Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Vedran Cirak

Mobilna aplikacija za podporo vzdrževalcem na terenu

diplomsko delo

visokošolski strokovni študijski program prve stopnje računalništvo in informatika

Mentor: doc. dr. Rok Rupnik

Ljubljana 2014

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Delo vzdrževalcev na terenu je tipičen primer poslovnega procesa, za potrebe katerega je uvedba mobilne aplikacije smiselna in celo potrebna. Analizirajte obstoječo mobilno aplikacijo za podporo dela vzdrževalcev na terenu, ki je razvita za platformo Windows Mobile. Na podlagi analize njenih funkcionalnosti razvijte mobilno aplikacijo za platformo Android, s čemer boste v bistvu prenovili staro mobilno aplikacijo. Nova mobilna aplikacija naj temelji na podatkih sistema Maximo, ki je vodilni sistem za podporo vzdrževanju na svetu.

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Vedran Cirak, z vpisno številko **63080390**, sem avtor diplomskega dela z naslovom:

Mobilna aplikacija za podporo vzdrževalcem na terenu

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom doc. dr. Rok Rupnika,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela,
- soglašam z javno objavo elektronske oblike diplomskega dela na svetovnem spletu preko univerzitetnega spletnega arhiva.

V Ljubljani, dne 14. oktobra 2014 Podpis avtorja:

Zahvaljujem se mentorju, doc. dr. Roku Rupniku ter sodelavcema Aleksandru Vrčku in mag. Janezu Hribarju za pomoč, nasvete in navodila pri izdelavi diplomske naloge. Le-te ne bi uspel narediti brez podpore svoje družine, ki mi je omogočila študij v Ljubljani. Zato se jim ob tej priložnosti iskreno zahvaljujem.

Za Svena in Eugen-Bernarda

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
PSA	Power Service Assistaint	PSA
WM	Windows Mobile	Windows Mobile
ND	Workorder	Nalog za delo
API	Application programming interface	Aplikacijski programski vmesnik
SDK	Software development kit	Komplet za razvoj prog. opreme
IDE	Integrated development kit	Integrirano razvojno okolje
GUI	Graphic User Interface	Grafični uporabniški vmesnik
MVC	Model View Controller	Model View Controller
DV	Transmission line	Daljnovod

Povzetek

Diplomsko delo opisuje programsko rešitev za vodenje in izvajanje nalog vzdrževanja na terenu. Delavcu se priredi nalog za delo, ki se prenese na napravo in izvede. Uporabnik opazuje in piše opombe o daljnovodih, nato aplikacija sinhronizira te podatke s strežnikom PSA, ki jih pridobiva in vrača iz sistema Maximo. Aplikacija PSAMobile je namenjena uporabi na mobilnih napravah, ki podpirajo operacijski sistem Android.

V prvem poglavju smo predstavili namen aplikacije PSAMobile ter opisali starejše, a še vedno funkcionalne verzije programa PSAMobile za OS Windows Mobile.

V drugem poglavju smo opisali uporabljene tehnologije, s katerimi je bila razvita aplikacija.

Tretje poglavje opisuje potek izdelave programske rešitve. Poudarek je na strukturi projekta PSAMobile v razvojnem okolju Eclipse, shranjevanju podatkov in uporabi Androidovih funkcij. Opisane bodo tudi funkcionalnosti, na katerih smo delali največ časa; postopek zagona aplikacije ter okna za prikaz podrobnosti naloga za delo.

V zadnjem poglavju smo predstavili uporabo aplikacije na konkretnih primerih s pomočjo zaslonskih slik.

Ključne besede: android, mobilne naprave, sinhronizacija podatkov.

Abstract

This thesis describes a programming solution for evidencing and performing work orders. The order is assigned to a maintainer, who executes it. Users (ie. maintenance worker) observe and note remarks about transmission lines to the work order, while the application synchronizes given data with the PSA main server. The server obtains and returns data in the Maximo system. PSAMobile is intended for mobile devices which support Android operating system.

In the first chapter we describe the purpose of PSAMobile. We will also take a brief look into the old version of PSAMobile for Windows Mobile OS.

The second chapter describes the technologies we used for development.

The third chapter describes development progress. Emphasis will be on the structure of the PSAMobile project in Eclipse IDE, storing data and usage of Android tools. We will also describe functionalities on which we have been working most of the time; the window for application initialization and for workorder details.

In the last chapter we will present example of application use on concrete cases with screenshots.

Keywords: android, mobile devices, data synchronization.

Poglavje 1

Uvod

1.1 Opis programa PSAMobile

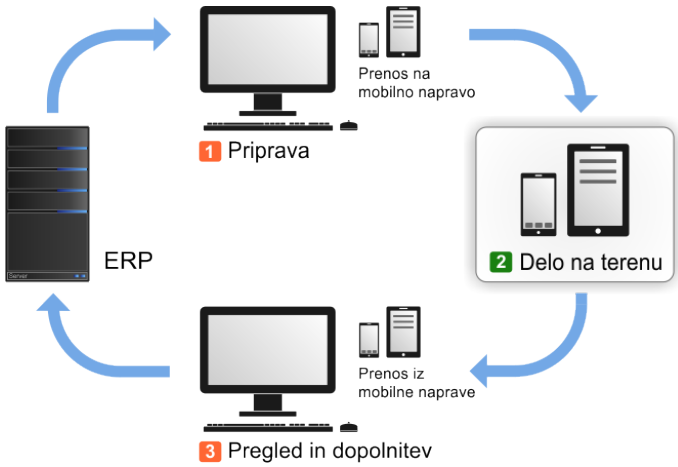
PSAMobile je program za mobilne naprave, s katerim se vodi in spremlja izvajanje nalog vzdrževanja na terenu. Mobilna naprava prejme podatke o nalogah s strežnika preko brezžične povezave. Strežnik podatke pridobiva (in vrača) iz sistema Maximo[1], nekatere podatke pa tudi iz drugih sistemov.

Priprava nalogov za izvajanje na terenu se pripravi v sistemu Maximo, kjer se nahajajo Nalogi za delo (ND). Pod ND se nahajajo konkretna opravila, med njimi tudi obhodna opravila.

Preden se podatki pošljejo na klienta oz. napravo se najprej v spletni aplikaciji PSA prevzamejo Nalogi za delo in opravila iz sistema Maximo. Potem se jih dopolni, po potrebi pa se določi/spremeni kdo in kdaj naj opravila izvaja. ND in opravila s fazo *Za teren* so označena, da se jih lahko prevzame na mobilne naprave, na katerih se izvaja aplikacija PSAMobile.

Ko terenski uporabnik izvede opravila, jih lahko označi. Lahko tudi doda novougotovljena opravila (ugotovoljene okvare), ki se dodajo izbranemu nadrejenemu Nalogu za delo. V kolikor ugotovitve še niso izvedene, potem po prenosu na strežnik ostanejo *nerazporejene*. Terenski uporabnik lahko na izbran Nalog za delo

[1]Maximo je IBM-ov sistem za upravljanje s sredstvi (angl. *Asset Manager*), v katerem se nahajajo podatki za naloge za delo, ki se ob sinhronizaciji pošiljajo na klienta.



Slika 1.1: Potek pridobivanja in pošiljanja podatkov.

ali opravila doda zapise o opravljenem delu in uporabljenih orodjih, ki se prenesejo nazaj na strežnik ob prenosu Naloga za delo oz. opravil.

Na strežnik v spletni aplikaciji PSA se vrnjene in novo opravljene naloge pregleda in nato prenese v Maximo ter izdela poročila. Nerazporejena opravila pa čakajo, dokler se jih ne razporedi na izbrane Naloge za delo in se jih prenese v Maximo, ali pa kar nazaj na teren v izvajanje. Potek pridobivanja in pošiljanja podatkov lahko razporedimo na naslednje korake:

- Priprava podatkov na spletni aplikaciji PSA iz Maximo sistema,
- Prevzem podatkov na mobilno napravo in
- Pošiljanje podatkov z mobilne naprave nazaj na strežnik (spletna aplikacija PSA).

1.2 Stara in nova aplikacija PSAMobile

Originalni mobilni sistem vzdrževanja PSAMobile deluje na principu zgornjega opisa, naša aplikacija za Android bo seveda za komunikacijo s strežnikom uporabljala tudi isti princip pridobivanja in pošiljanja podatkov. Stara aplikacija je bila razvita v okolju Visual Studio in je podprta za OS Windows Mobile, ki za komunikacijo s strežnikom uporablja knjižnici `LibPsaComData` in `PsaComData`.

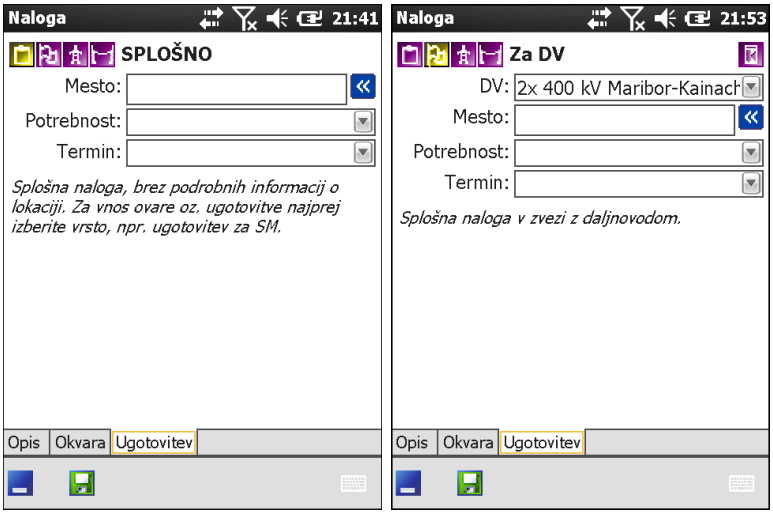
Ob prvem zagonu aplikacije se prikaže prijavno okno (Slika 1.2) za vnos uporabniškega imena in gesla. Ko uspešno vnesemo podatke, se nam prikaže glavno okno, ki vsebuje krovne Naloge za delo(ND[2]).

Vsak krovni ND ima svoje otroke oz. terenske naloge, ki spet lahko vsebujejo svoje otroke oz. ugotovitve (4 vrste) in obhode. Glede na izbran ND ali njegovega otroka se nam prikažejo tudi ustrezne opcije, kot so na primer podrobnosti o izbranem ND, terenski nalogi ali ugotovitvi, dodajanje nove terenske naloge ali dodajanje ene od 4 možnih ugotovitev. Za vsako terensko nalogo in ugotovitev lahko dodamo tudi seznam *Opravljenega dela* in seznam *Orodij*.

Pomembno dejstvo pa je, da nekatere ND ne moremo urejati, saj obstajajo določene restrikcije. Tako so npr. **vsi krovni ND, pridobljeni iz Maximo strežnika, namenjeni le branju podatkov**, ne pa tudi spreminjanju.

Slika 1.2: Stara mobilna aplikacija - Prijavno okno.

[2]Od sedaj naprej bomo za Nalog za delo uporabljali kratico ND



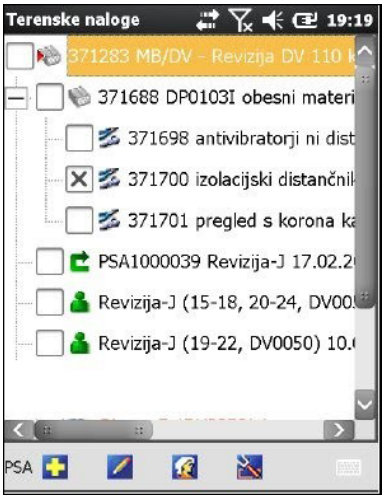
Slika 1.3: Stara mobilna aplikacija - Primer ugotovitvah.

1.2.1 Hierarhični seznam

Med razvojem je bil star program podlaga za razvoj aplikacije za OS Android. Podatkovna struktura je bila nadgrajena iz stare aplikacije, vzporedno pa je stara bila dograjena za podporo novih elementov v modelu. Največ dela pa je bilo pri Graphic User Interface-u in sicer pri izdelavi hierarhičnega seznama, ki se bistveno razlikuje od večnivojskega drevesnega seznama (Slika 1.4) iz starega programa PSAMobile.

Android namreč nima uradni widget[3] za prikaz večnivojskega drevesnega seznama kot v WM-ju. Pri WM je prikaz le-tega enostaven, saj imamo lahko na enem mestu vpogled v vse otroke od izbranega ND. Ker ima torej vsak krovni ND lahko tudi do pet otrok, je bilo potrebno razviti lasten hierarhični seznam za OS Android. Poudarek je bil predvsem na učinkovitosti in prilagodljivosti za vse njegove verzije (od verzije 4.0 naprej).

[3]Widget je enostavna komponenta ali aplikacija namenjena za eno ali več softverskih platform.



Slika 1.4: Stara mobilna aplikacija - Drevesni seznam.

1.3 Struktura aplikacije PSAMobile

Na kratko bomo za lažjo predstavitev opisali strukturo aplikacije PSAMobile oz. njene funkcije in gradnike. Vsa našteta okna so narejena iz t. i. *Activity*-ja, ki znotraj sebe lahko vsebuje še `ViewPager` s fragmenti. Pomen *Activity*-ja in *Fragment*-a bomo spoznali v naslednjem poglavju.

- **Login** - Activity za avtentikacijo,
- **Pozdravno okno** - Ob kliku na gumb *Prijava* se nam prikaže *Progress bar*,
- **Glavno okno oz. drevesni seznam** - ViewPager v kombinaciji s fragmenti, ki lahko vsebuje:
  - Nalog za delo,
  - Terensko nalogo in
  - Ugotovitev, ki je lahko:
    - Splošno,
    - Daljnovod,
    - Za stojno mesto in
    - Razpetina.
- **Podrobnosti za ND, terensko nalogo ali opravilo** - ViewPager, vsebuje tri zavihke oz. fragmente:
  - Opis,
  - Okvara - vsebuje poseben Activity ***Klasifikacija okvar*** in

  - Ugotovitev - vsebuje Activity ***Tehnični podatki*** in ***Zemljevid***.
- **Opravljeno delo - Seznam delavcev in količina opravljenega dela**,
- **Izbira orodja - Seznam orodij**,
- **Izmenjava in podatki** - ViewPager s štirimi fragmenti:
  - Uvod - osnovni podatki o uporabniku,
  - Izmenjava - pridobitev podatkovne baze s strežnika,
  - Odstrani - odstrani tekočo podatkovno bazo in
  - Obnovi - obnovi podatkovno bazo iz arhiva ali kopije.
- **Sistemske nastavitve** - ViewPager s tremi fragmenti:
  - Uvod,
  - Nastavitve in
  - Test.
- **Nastavitve strežnika** - Nastavitve za strežnik,
- **Nastavitve zemljevidov** - Nastavitev za pot do zemljevidov,
- **Snemalnik poti** - GPS snemanje poti,
- **O aplikaciji**.

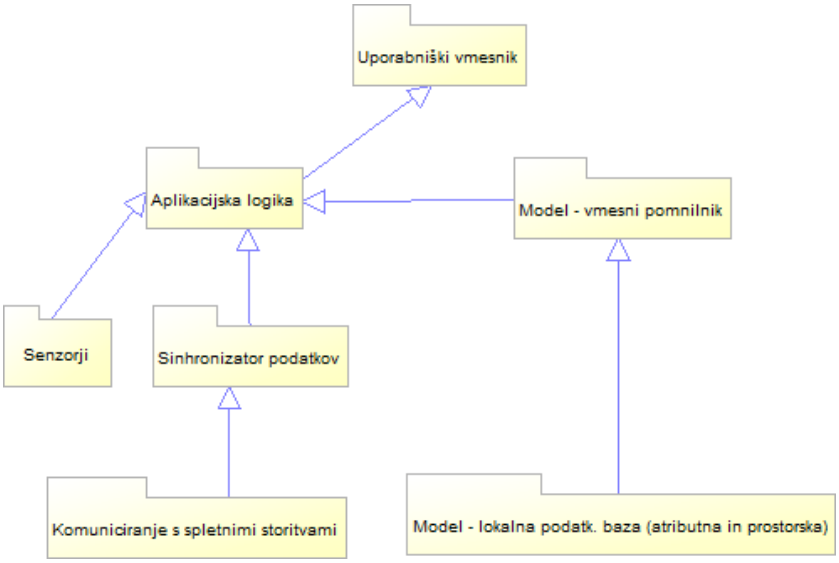
1.4 Podatkovni modeli

Struktura mobilne aplikacije (Slika 1.5.) sledi konceptu MVC:

- *Model* - model podatkov za prikaz ter podatkov za hranjenje v lokalni bazi,
- *View* - pogled na podatke (uporabniški vmesnik) za predstavitev uporabniku,
- *Controller* - povezovalna in aplikacijska logika.

Naš *Controller* vsebuje še dva pomembnejša sklopa:

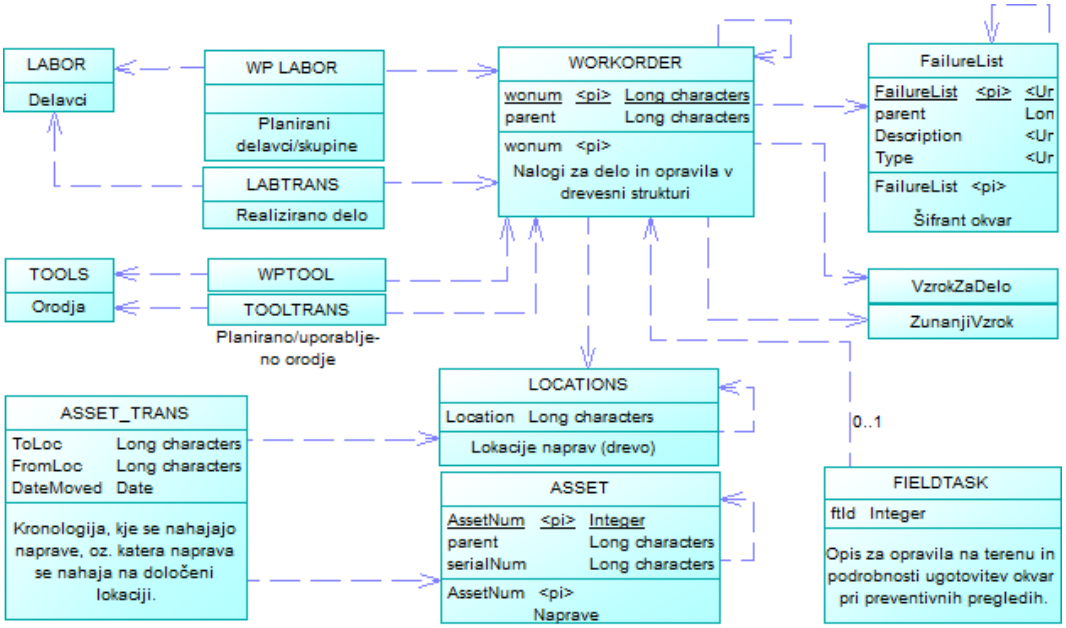
- **Senzorji** - logika zajemanja GPS, orientacije oz. rotacije naprave ter kompasa za podporo prikaza zemljevidov,
- **Sinhronizator** - komunikacijska logika za učinkovito prenašanje podatkov, obravnavo konfliktov ter upravljanje s pomnilnikom.



Slika 1.5: Zgradba aplikacije.

Na naslednji sliki (Slika 1.6.) je prikazan poenostavljen model podatkovne baze, kjer so navedeni najbolj pomembni šifranti:

- **Workorder** - nalogi za delo,
- **Fieldtask** - opravila na terenu, opisujejo okvare pri preventivnih pregledih,
- **Naprave** - šifrant *Asset*,
- **Delavci in orodja** - šifranta *labor* in *tools*,
- **Šifranti za opis okvar** - drevesna organizacija šifranta okvare (Napaka, problem, vzrok in odprava).



Slika 1.6: Diagram podatkovne baze.

Poglavje 2

Pregled uporabljenih tehnologij za razvoj programske rešitve

V nadaljnih poglavjih bomo spoznali tehnologije, ki smo jih uporabljali pri izdelavi aplikacije PSAMobile. Opisali bomo tudi OS Android in njegove najbolj pogosto uporabljene razrede ter *widgete*.

2.1 Android

Android je Googlov operacijski sistem na osnovi Linux jedra, prilagojen predvsem za mobilne naprave na dotik, kot npr. pametne telefone ter tablične računalnike.

Android Inc. so oktobra 2003 ustanovili Andy Rubina, Rich Miller, Nick Sears in Chris White s ciljem razvoja aplikacij za pametne mobilne naprave. Leta 2005 je Google prevzel Android in začel boj z drugimi mobilnimi platformami na trgu, kot so iOS ali Windows Phone. Leta 2008 je izšla prva komercialna verzija, ki ni imela kodnega imena. Zahvaljujoč odprtokodnosti se je v zelo kratkem času razširil na trgu mobilnih naprav in je danes eden glavnih v svetu mobilnih operacijskih sistemov. Po poročilih analitske hiše Strategy Analytics je bil od januarja do novembra 2013 Android nameščen na osmih od desetih prodanih pametnih telefonih [1].

Celotna izvorna koda operacijskega sistema Android je dostopna pod Apache [3] licenco. Danes je lahko katerakoli mobilna naprava opremljena z OS Android, vendar mora biti v skladu s pogoji, definiranimi v Compatibility Definition

Document (CDD)[1].

Interakcija z uporabnikom je omogočena s pomočjo zaslona, ki je občutljiv na dotik (t. i. *touch screen*), s katerim lahko upravljamo z objekti na zaslonu.

2.1.1 Platformske različice

Skozi leta je Google razvil več različic OS Android-a. Vsaka je dobila ime po znanih slaščicah, in sicer po vrstnem abecednem redu. Prva konkretna verzija se pojavlja pod imenom Android 1.5 (angl. *Cupcake*), nato so sledile še:

- **1.6** *Donut*,
- **2.0** *Eclair*,
- **2.2** *Froyo*,
- **2.3** *Gingerbread*,
- **3.0** *Honeycomb*,
- **4.0** *Ice Cream Sandwich*,
- **4.1/4.2/4.3** *Jelly Bean* in
- **4.4** *Kit Kat*[2].

2.1.2 Java in Android SDK

Aplikacije, namenjene operacijskemu sistemu Android, se razvijajo v programskem jeziku Java z uporabom Android Software Development Kit-a (SDK). SDK vsebuje orodja za razvoj, kot so razhroščevalnik, programske knjižnice, emulator, dokumentacija in primeri s kodo. Uradno podprto okolje za razvoj je Eclipse IDE. Android ponuja možnost pridobitve ADT Bundle-a na uradni spletni strani, ki vsebuje Android SDK komponente ter razvojno okolje Eclipse z vgrajenim ADT-jem. Na ta način dobimo vse v enem kosu in smo pripravljeni za razvoj programske opreme.

Google bo v prihodnosti podpiral novo razvojno okolje Android Studio, ki temelji na IntelliJ IDEA [2] z integriranim ADT-jem.

[1] Več informacij lahko najdemo na http://source.android.com/compatibility/.
[2] Zadnja različica, ki je izšla 14. oktobra 2013

2.1.3 Activity

Aktivnost (angl. *Activity*) je aplikacijska komponenta, ki ponuja zaslon, preko katerega lahko uporabnik opravlja določena opravila, kot npr. kliče v mobilna omrežja, pošilja e-pošto ali slika fotografije. Aktivnost uporablja datoteko *xml*, ki določi izgled grafičnega uporabniškega vmesnika. Android aplikacija lahko vsebuje več aktivnosti, ki so povezane med seboj. Zaženemo jo s pomočjo razreda `Intent`, v katerega lahko spravimo argumente in jih uporabimo v naslednji aktivnosti. Ob vsakem zagonu aktivnosti se prejšnja ustavi, operacijski sistem pa jo ohranja v t. i. kopici oziroma *back stack*-u, ki temelji na mehanizmu *last in, first out*. Tako npr. uporabnik lahko pritisne gumb *Back* in se trenutni Activity odstrani iz kopice, prejšnji pa se ponovno prikaže. Activity lahko zaženemo na dva načina, in sicer s pomočjo metod:

- `startActivity(Intent intent)` in
- `startActivityForResult()`.

Prva metoda `startActivity(Intent intent)` zažene novo aktivnost, ki nam ne vrača rezultata. To pomeni, da ne bomo vedeli, da se je aktivnost končala. Uporabljamo jo torej takrat, kadar vemo, da ne bomo potrebovali rezultata iz aktivnosti, ki jo bomo zagnali. Ob kreaciji objekta `Intent` kot prvi argument navedemo trenutno aktivnost, kot drugi pa aktivnost, ki bi jo radi zagnali (Koda 1.).

```
Intent intent = new Intent(this, NextActivity.class);
intent.putExtra("stringKey", "stringValue");
intent.startActivity(intent);
```

Koda 1: Zagon aktivnosti brez rezultata.

Druga metoda `startActivityForResult(Intent intent, int reqCd)` nam za razliko od prve vrne rezultat. Namen je torej popolnoma isti, le da trenutna aktivnost pričakuje rezultat po koncu nove aktivnosti. Drugi argument služi za kasnejše primerjanje v metodi `onActivityResult()` trenutne aktivnosti (Koda 2.).

Tako smo v naši aplikaciji imeli dodajanje opravljenega dela (`EditLabor Activity`), iz katerega smo zagnali novo aktivnost `ChooseWorkerActivity`, v

```
// EditLaborActivity
Intent intent = new Intent(this, ChooseWorkerActivity.class);
startActivityForResult(chooseLaborIntent, REQUEST_CODE);
```

Koda 2: Zagon aktivnosti z rezultatom.

kateri smo iz seznama delavcev izbrali enega, končali aktivnost in rezultat (ID izbranega delavca) pripeljali nazaj v aktivnost za opravljeno delo. V metodi `onActivityResult()` pa naprej delamo s pridobljenimi rezultati. V našem primeru smo na podlagi ID delavca prikazali njegovo ime in priimek v UI kontroli `Spinner` (Koda 3.).

```
// EditLaborActivity
if (resultCode == RESULT_OK && requestCode == REQUEST_CODE){
 if (data.hasExtra(parChoosenLaborCode)) {
  String code = data.getStringExtra(parChoosenLaborCode);
  labtrans.setLaborcode(code);
  selectSpinnerItem(code);
 }
}
```

Koda 3: Metoda *onActivityResult()*.

Vsaka aktivnost ima svoj življenski cikel. Aktivnost se začne v metodi `onCreate()`, kjer se najprej naloži GUI iz datoteke *xml*. Nalaganje te datoteke pomeni, da se koda xml oz. njeni elementi spremenijo v t. i. *živo* kodo, s katero nato programsko upravljamo. Tako do GUI elementa TextView dostopamo s pomočjo metode `findViewById(R.id)`, ki ji ko argument podamo ID, ki smo ga definirali v datoteki xml. Aktivnost bo v `onCreate()` (Koda 4.) metodi inicializirala tudi njeno *globalno* stanje, npr. izvajanje določene niti (angl. *thread*-a). `onStart()` se pokliče, ko `Activity` uporabniku postane viden, prikaže se torej njegov UI. `onResume()` je zadnja metoda v procesu izgradnje `Activity`-ja in se pokliče ob začetku interakcije aktivnosti z uporabnikom.

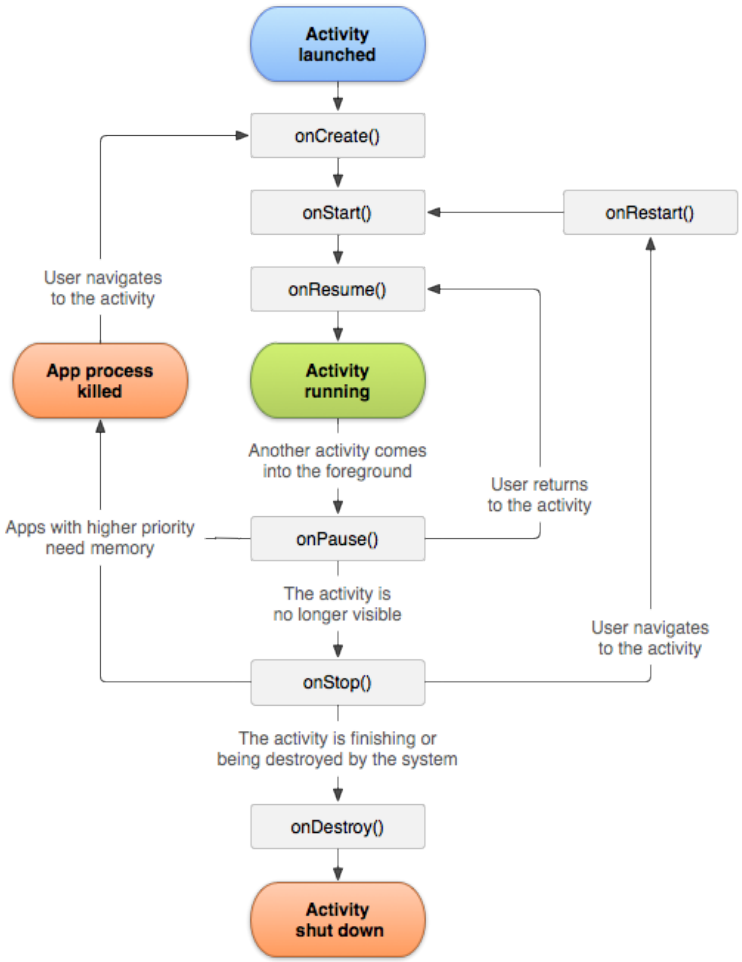
V kolikor uporabnik zapre `Activity` z določeno akcijo, kot je npr. prehod v prejšnjo aktivnost, se najprej pokliče metoda `onPause()`. Tukaj običajno shranjujemo podatke v persistenco, ustavimo animacije in druge stvari, ki uporabljajo procesorski čas. Predzadnja metoda, ki se pokliče ob zapiranju aktivnosti je `onStop()`,

```java
@Override
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);

  // nalozimo xml datoteko za GUI
  setContentView(R.layout.main_activity);

  // inicializiramo element TextView
  mTextView = (TextView) findViewById(R.id.text_message);
}
```

Koda 4: Inicializacija aktivnosti v metodi *onCreate()*.

pri kateri UI uporabniku ni več viden, saj ga že pokriva nov `Activity`.

Aktivnost se konča v metodi `onDestroy()`, kjer se sprostijo tudi drugi viri (kot omenjene niti v metodi `onCreate()`). Je končna metoda, preden se `Activity` uniči. Zgodi se, če nekdo pokliče metodo `finish()` ali pa operacijski sistem uniči aktivnost z namenom prihranka prostora v pomnilniku. Celoten potek življenjskega cikla aktivnosti lahko vidimo na Sliki 2.1.

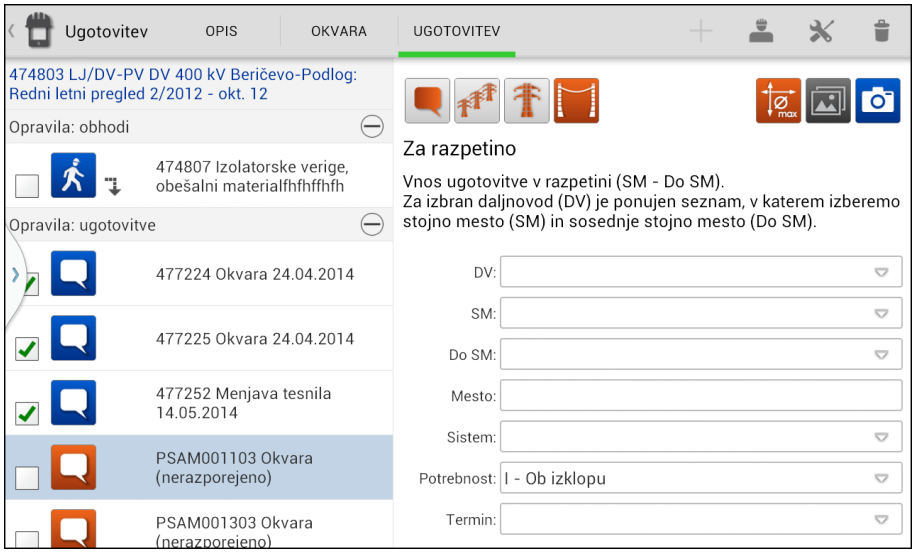


Slika 2.1: Življenski cikel aktivnosti.

2.1.4 Fragment

`Fragment` predstavlja obnašanje ali del uporabniškega vmesnika v `Activity`-ju. Eno aktivnost lahko kombiniramo z več fragmenti [6].

Tako smo v naši programski rešitvi za tablične računalnike uporabili t.i. *master-detail* pogled. Omenjeni pogled je sestavljen iz ene aktivnosti, ki ima v primeru horizontalne lege na levi strani drevesni seznam (prvi fragment), na desni pa podrobnosti o ND ali ugotovitvi (drugi fragment). V kolikor tablični računalnik obrnemo v vertikalno pozicijo, se nam prikaže le prvi fragment znotraj iste aktivnosti, drugi pa se uniči.



Slika 2.2: Master-detail prikaz za 10 inčne tablete.

Tudi fragment ima svoj življenjski cikel, ki je odvisen od življenjskega cikla `Activity`-ja, zato mora vedno biti del aktivnosti. Ko npr. pavziramo ali uničimo `Activity`, se ustavijo/uničijo tudi vsi njegovi fragmenti. `Activity` lahko manipulira s fragmentom tako, da ga doda ali odstrani iz njegove kopice. Poleg standardnih življenjskih metod `Activity`-ja vsebuje `Fragment` še naslednje metode:

- `onAttach()` - prva metoda, ki se pokliče ob kreaciji fragmenta. V njem nastavimo t.i. *callback*-e za komunikacijo med fragmentom in `Activity`-jem,
- `onCreateView()` - pokliče se, ko je čas za izris uporabniškega vmesnika. Da bi do tega prišlo moramo vedno s pomočjo `super` metode vrniti `View` od korenskega `layout`-a,
- `onActivityCreated()` - metoda, ki indicira, da je aktivnost kreirana,
- `onDestroyView()` - klic metode, ko je `View` pripravljen za odstranjevanje iz korenskega `layout`-a in
- `onDetach()` - zadnja metoda v življenjskem ciklu fragmenta, v kateri lahko uničimo instanco `Callback`-a, saj komunikacija z aktivnostjo v tem trenutku ni več potrebna.

Fragment živi v objektu `ViewGroup` kot del *layout* aktivnosti. Lahko ga dodamo neposredno v *xml layout* z značko `<fragment>` ali programsko v objekt `ViewGroup` s pomočjo objekta `FragmentTransaction` kot v naslednjem primeru:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    FrameLayout frame = new FrameLayout(this);

    if (savedInstanceState == null) {
        mFragment = new ExampleTwoFragment();
        FragmentTransaction ft =
                getFragmentManager().beginTransaction();
        ft.add(frame.getId(), mFragment).commit();
    }
    setContentView(frame);
}
```

Koda 5: Nalaganje fragmenta.

Toda poudariti je treba, da za fragment ni nujno, da je vedno del `layout` aktivnosti. Fragment lahko uporabljamo brez uporabe njegovega grafičnega vmesnika oz. kot *delavca* (angl. *worker*) za `Activity`.

2.2 Eclipse

Za razvoj aplikacije PSAMobile smo uporabljali Eclipse razvojno okolje, ki je v kombinaciji z ADT-jem občasno nezanesljiv. Namreč: vsako toliko ga je bilo treba po nekaj urah dela znova zagnati, saj bi naenkrat izpisal poljubno napako na samem projektu. Če tega ne bi storili, ne bi mogli zagnati aplikacije za postopek razhroščevanja.

Za delo z GUI-jem Eclipse nudi grafični urejevalnik, toda večino časa smo razvijali v *xml* urejevalniku. Emulatorji se definirajo v programu *Android Virtual Device Manager*. Postopek debugiranja v emulatorjih je sicer zelo počasen, drugi primanjkljaj je dejstvo, da ne delamo na fizični napravi, kjer so rezultati lahko bistveno drugačni kot na emulatorju. Zato smo za razvoj in debugiranje uporabljali različne tablične računalnike oz. fizične naprave.

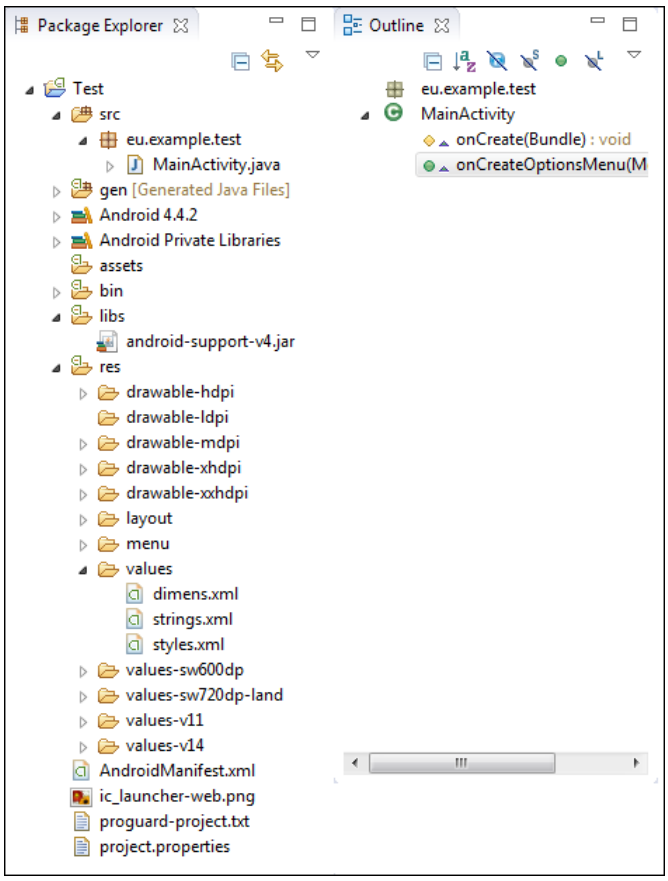
2.2.1 Perspektive

Java - najbolj pogosto uporabljena perspektiva, kjer večino prostora zasega okno za pisanje izvorne kode. Na levi strani imamo prikazan *Package Explorer* za lažji

vpogled v strukturo projekta. Pomembne mape:

- **src** - paketi, ki vsebujejo razrede Jave,
- **assets** - mapa za odlaganje datotek, iz katerih jemljemo podatke (npr. podatkovna baza),
- **libs** - mapa za knjižnice,
- **res** - vsebuje vire za grafični prikaz (npr. slike), nastavitve za različne naprave in druge *xml* datoteke:
  - **drawable** - mapa za odlaganje slik, ločimo jo glede na resolucijo (ldpi, hdpi, xhdpi in drugo),
  - **layout** - mapa za *xml* datoteke, ki definirajo GUI,
  - **menu** - mapa za definiranje ActionBar menija s pomočjo *xml* datotek,
  - **values** - vsebuje *xml* datoteke za dostop do nizov, dimenzij (glede na vrsto naprave) ali do stilov.

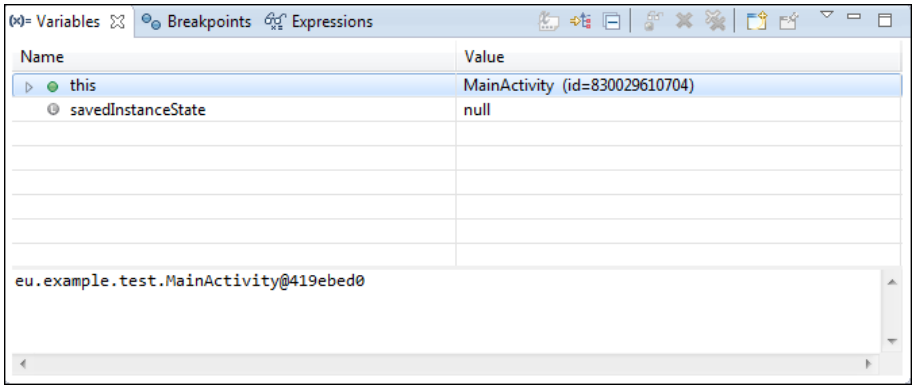
Na desni strani se nahaja okno *Outline* za vpogled v napisane metode trenutnega razreda. Zelo koristno okno, v kolikor razred vsebuje večjo količino napisanih metod. Zgled lahko vidimo na naslednji strani (Slika 2.3.)



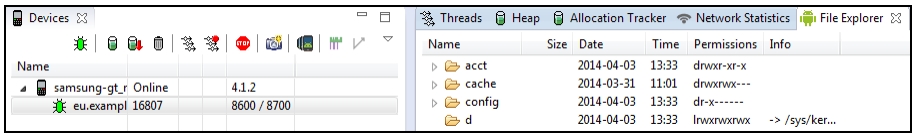
Slika 2.3: *Package Explorer* in *Outline*.

Debug - Debug je druga najbolj uporabljena perspektiva, kjer smo v glavnem imeli odprto okno za pisanje izvorne kode (zaradi breakpointov) ter zavihka *Variables* in *Expressions* za vpogled v trenutne instance razredov in njihove vrednosti (Slika 2.4.). Razlika med zadnjima dvema oknoma je ta, da so v oknu *Variables* prikazane privzete instance razredov. V oknu *Expressions* pa so prikazane instance in spremenljivke, ki jih je programer dodajal med razhroščevanjem s pomočju ukaza *Watch*.

DDMS - skrajšava za Dalvik Debug Monitor Server. Orodje za debugiranje naprave omogoča prikaz datotečnega sistema na napravi, vpogled v procese in niti, informacije o napravi ter zajem zaslonske slike (Slika 2.5.).



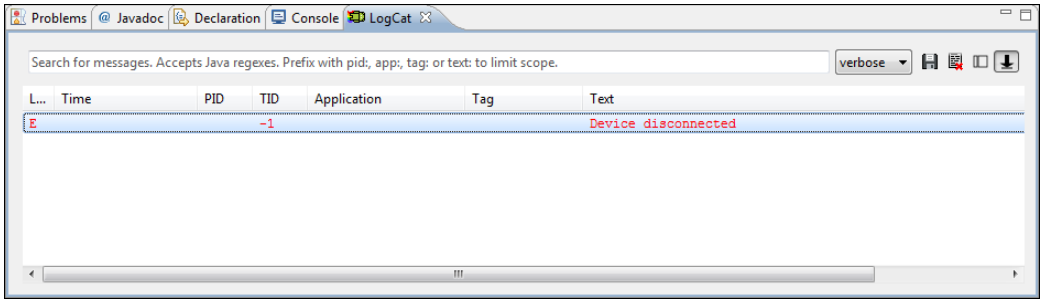
Slika 2.4: Zavihki *Variables*, *Breakpoints* in *Expressions*.



Slika 2.5: DDMS - seznam naprav in prikaz datotečnega sistema.

2.2.2 LogCat

Ena od pomembnih zadev v Eclipse ADT-ju je bil tudi **LogCat**. LogCat omogoča vpogled v celoten *log* neke naprave. Tako smo pri določenih težavah postavljali loge v kodo, nato zagnali aplikacijo v *debug mode* in spremljali izpis rezultatov v okno (Slika 2.6).



Slika 2.6: Logcat okno v Eclipse-u.

2.2.3 SVN

SVN [4] je centralizirani in brezplačni sistem za upravljanje z izvorno kodo, ki je nastal leta 2000 kot alternativa CVS-ju[3]. Uporablja se za pridobitev trenutnih in zgodovinskih datotek, kot so izvorna koda, spletne strani in dokumentacija [5].

Glavne operacije:

- *Commit* - Atomična operacija s katero oddamo lokalno spremenjene podatke na strežnik,
- *Update* - S strežnika dobimo najnovejšo verzijo, ki so jo drugi razvijalci prej shranili. Operacija spremeni lokalno strukturo podatkov,
- *Checkout* - Uporabljamo kadar v *Workspac* želimo dodati obstoječi projekt iz verzijskega sistema.

Glavni projekt se nahaja v t. i. *deblu* (angl. *trunk*), vse kopije projekta pa držimo v t. i. *veji* (angl. *branch*). Vejo bomo naredili takrat, ko se izkaže potreba za istim projektom, in sicer s funkcionalnostmi, ki se ne nahajajo v glavnem projektu oz. deblu. Na tak način glavni projekt ostane stabilen, v vejah pa pišemo nove funkcionalnosti in jih testiramo. Ko so nove funkcionalnosti pripravljene in smo prepričani, da so tudi stabilne, potem lahko naredimo t. i. *združevanje* (angl. *merge*) veje in debla. S pomočjo te operacije prenesmo vse nove funkcionalnosti v glavni projekt.

2.3 Knjižnice

Za uspešno komunikacijo s strežnikom in pridobivanje podatkov smo uporabljali knjižnice, ki so bile že prej razvite za potrebe operacijskega sistema Windows Mobile. Omenjene knjižnice je vodja projekta prilagodil Javi, saj so originalne napisane v programskemu jeziku C#. Zelo prav so nam prišle tudi javne knjižnice, ki so nam olajšale delo pri logiranju napak ter knjižnice za serializacijo in deserializacijo podatkov. V nadaljevanju bomo našteli, nato pa opisali najbolj pomembne knjižnice.

[3]CVS (Concurrent Versions System) je sistem za upravljanje z izvorno kodo, nastal leta 1990. Več informacij lahko dobimo na naslednjem spletnem naslovu: http://en.wikipedia.org/wiki/Concurrent_Versions_System.

- Knjižnice, razvite v podjetju:
  - LibPsaComData,
  - LibPsaData,
  - LibUtil in
  - LibPsaGis.
- Javno dostopno knjižnice:
  - Jackson,
  - Joda time in
  - Slf4j.

2.3.1 Knjižnice razvite v podjetju

LibPsaComData je knjižnica komunikacijskih objektov, ki jih uporablja protokol komunikacije med klientom in strežnikom, torej razredi, ki predstavljajo podatke zahtev strežniku, ter razredi, ki predstavljajo odgovore strežnika, ter seveda podatkovni razredi, ki jih zahteve in odgovori vsebujejo, npr. razred `Mv_workorder_v`, ki predstavlja posamezno ND/Opravilo. Razredi knjižnice so torej poznani tako klientu kot strežniku in predstavljajo *dogovor o slovarju besed*, ki jih uporabljata v komunikaciji.

LibPsaData je knjižnica, ki jo pozna le klient. Vsebuje podatkovni model klienta, tj. vse podatke, nad katerimi klient izvaja svoje operacije. V aplikaciji je javno poznan le majhen del knjižnice (*data engine*), preko katerega aplikacija dostopa do podatkov in nove/spremenjene shranjuje.

Pomembni notranji gradniki knjižnice pa so:

- syncer - podsistem za komunikacijo s strežnikom, ki poskrbi, da klient razpolaga z vsemi potrebnimi podatki (torej vsak podatek, ki ga klient ne potrebuje več, ga odstrani, da je lokalna baza podatkov kar najmanjša), in seveda poskrbi, da se na strežnik prenesejo podatki, pripravljeni za prenos na strežnik,
- persister -podsistem za branje in hranjenje podatkov v lokalni podatkovni bazi. Podsistem je optimiziran za mobilno napravo (čim manjša poraba virov, hitra odzivnost, po drugi strani pa za optimizacijo omejena funkcionalnost).

LibUtil - knjižnica, ki vsebuje različne razrede značilne za več Activity-jev. Je zelo pogosto uporabljana knjižnica, saj nam ni bilo treba za akcije, ki se ponavljajo, pisati podvojene kode. Tako smo na primer za sinhronizacijo podatkov s strežnikom in logiranje uporabili razred `SimpleLoaderManager`.

LibPsaGis - GIS[4] knjižnica za prikaz zemljevidov, ki vsebuje več vrst map. Do map dostopamo s pomočjo ESRI[5] API-ja in sicer v *offline* ali *online* načinu izvajanja. Uporabniku na zemljevidu pomaga najti trenutno pozicijo (označeno s točko) stojnega mesta ali daljnovoda.

V spodnjem seznamu bomo našteli tudi druge pomembne razrede iz knjižnice LibUtil, ki so se uporabljali pri izdelavi aplikacije PsaMobile:

- `TreeActivity` - generična aktivnosti, ki smo jo uporabili pri izdelavi glavnega drevesnega seznama in seznama za izbiro klasifikacijskih napak,
- `DateFormatter` - za pretvorbo datuma v ISO-8601 datumski čas, saj strežnik za komunikacijo upošteva le datume v tej obliki,
- `SerializeTool` - orodje za serializacijo in deserializacijo objektov. Prav nam je prišlo, ko smo serializirali/deserializirali preproste objekte. Bolj kompliciran razredi so implementirali vmesnik Serializable ali Parcelable,
- `SimpleLoaderManager` - vsebuje anonimni notranji razred Loader z dvemi metodami; `onStartLoading()` in `loadInBackground()`. Prva metoda nam omogoča upravljanje z instancami, ki se še nahajajo v GUI niti (npr. Spinner, ProgressBar). Druga pa sproži delovno nit za izmenjavo podatkov s strežnikom. `SimpleLoaderManager` vsebuje še metodo `onLoadFinished()`, ki se aktivira po metodi `loadInBackground()`. `onLoadFinished()` je metoda, ki se izvaja v GUI niti in v njej uporabniku posredujemo končni rezultat sinhronizacije, ki je lahko uspešen ali neuspešen,
- `Formaters` - razred z metodami za pretvorbo iz različnih tipov kot so `Double`, `Integer`, `Date` v formatiran String objekt,
- `TableListView` - razširja razred `FrameLayout` in ga uporabljamo pri izbiri delavcev in orodij. Omogoča tabelarični prikaz podatkov v aktivnosti.

[4]Geographic Information System je sistem za zajemanje, manipuliranje in analiziranje vseh tipov geografskih podatkov.

[5]ESRI je internacionalni dobavitelj GIS software-a, web GIS-a in aplikacij za upravljanje z geografskimi podatki.

2.3.2 Javno dostopne knjižnice

Jackson - Spada med najhitrejše JSON parserje, omogoča zasebno konfiguracijo ter popolno povezovanje podatkov z Java bean razredi, zbirkami (angl. *Collections*), različnimi podatkovnimi strukturami (kot je npr. `Map`) in `enum`-i. Za serializacijo in deserializacijo (Koda 7. in Koda 8.) podatkov nam služi objekt `ObjectMapper`. Z njegovo instanco nastavljamo vse možne lastnosti za obdelavo podatkov:

```
ObjectMapper m = new ObjectMapper();
m.configure(DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES, false);
```

Koda 6: Nameščanje *JSON* lastnosti.

```
mapper.writeValueAsString(conRequest);
```

Koda 7: Serializacija.

```
mapper.readValue(jsonString, TestDataResponse.class);
```

Koda 8: Deserializacija.

Joda time - Knjižnica, ki zamenja privzeto knjižnico `java.util.Date`. Čas je zapisan v ISO8601 standardu. Uporabljamo jo kot pripomoček, saj `java.util.Date` ne omogoča branje časa z odmikom (angl. *offset*-om, ki je razdeljen z dvopičjem (...+02:00).

Slf4j - Knjižnica za izpis *log* sporočil. Deklarirali smo jo v vsaki aktivnosti kot statično finalno spremenljivko. V kolikor med delovanjem aplikacije pride do napake, jo s pomočjo slf4j logiramo. Same napake se beležijo tudi v Android *log* sistemu, v katerega imamo vpogled s pomočjo *LogCat*-a v Eclipse-u. Obstaja več vrst napak in jih delimo glede na vrsto težave:

- **warn** - napaka, ki ne bo povzročila težave pri delu z aplikacijo, toda vseeno jo je treba odpraviti,

- **error** - napaka, ki v večini primerov povzroči sesutje aplikacije. Ponavadi so to napake tipa `NullPointerException`, `OutOfBoundsException`, `OutOfMemoryException`,
- **debug** - služi za izpis informacij. Ni nujno napaka,
- **info** - tudi za izpis informacij razvijalcem, npr. vzpostavljena povezava s strežnikom ali HTTP odgovor strežnika.

Poglavje 3

Izdelava programske rešitve

Za izdelavo programske rešitve smo morali najprej postaviti osnovne temelje in zbrati zahteve stranke. Aplikacija se bo uporabljala na mobilnih in tabličnih napravah, ki podpirajo OS Android verzije 4.0. Na samem začetku smo izgled aplikacije risali ročno na kosu papirja. Uporabniški vmesnik smo dizajnirali na najbolj enostaven način. Aplikacija se namreč uporablja na odprtem prostoru in je za pregled/dodajanje ND ali ugotovitve najbolj pomembna praktičnost in enostavnost vmesnika. S konkretno izdelavo rešitve smo začeli po 2 mesecih priprave in planiranja.

3.1 Organizacija

Vse, kar je vezano na projekt, smo beležili na spletu *Google Sites*, ki služi za enostavno organizacijo in dokumentacijo projektov [7]. Tako je kdorkoli od razvijalcev lahko dostopal do potrebnih informacij na enem mestu. Za vsak sklop projekta smo imeli posebno stran, kjer so bile informacije v zvezi z načinom implementacije rešitve in morebitni problemi ob njeni realizaciji.

Projekti (glavni projekt in knjižnice) so poimenovani glede na vsebino. Poimenovanje projekta je praviloma izpeljano iz naziva *nosilnega* paketa projekta, kjer izpustimo `eu.miga` prefiks, namesto pik pa uporabimo velike črke za separator. Tako npr. nosilni paket `eu.miga.lib.util` daje naziv projekta **LibUtil**.

3.2 GUI

Oblika uporabniškega vmesnika v OS Androidu je lahko shranjena v *xml* datotekah. Vsaka aktivnost v metodi ob njeni kreaciji z metodo `setContentView(R.layout.datoteka.xml)` celoten xml pretvori v t. i. živo kodo in ga naloži kot del uporabniškega vmesnika. GUI lahko gradimo bodisi z xml značkami bodisi programsko. V našem projektu smo ga sestavljali večinoma v datotekah xml in, če je bilo potrebno, smo med izvajanjem aplikacije določene dele vmesnika programsko nastavljali (npr. animacije ob kliku). Instanco grafičenga elementa iz datoteke programsko dobimo s pomočjo metode `findViewBy()` (Koda 9.). Vsak

```
// Vsak graficni element mora vsebovati ID
EditText editText = (EditText) findViewById(R.id.editTextId);
```

Koda 9: Primer metode *findViewById()*.

GUI element (oz. značka *xml*) vsebuje določene parametre s katerimi ga lahko pozicioniramo (glede na postavitev *layouta*), mu menjamo širino in višino, določamo barvo ali ga skrivamo/prikazujemo:

```
<TextView
    android:id="@+id/txtBottomDescription"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_weight="1"
    android:gravity="bottom"
    android:text="@string/login_activity_txt_btm_desc"
    android:textSize="@dimen/text_size" />
```

Koda 10: Značka ali grafični element *TextView*.

Najbolj pomembno je pozicioniranje elementa znotraj *layout*-a. Največ časa smo se srečevali z dvema osnovnima razporedoma; `LinearLayout` [16] ter `RelativeLayout` [17], ki izhajata iz korenskega elementa `View`. Oba elemeta sta tudi korenska elementa datoteke *xml*.

Prvi linearno razporeja njegove otroke v vrstici ali stolpcu. Orientacijo, ki je lahko vertikalna ali horizontalna, določamo z lastnostjo `android:orientation`.

Včasih je bila tudi lastnost `weight` zelo uporabna, saj nam je omogočala, da element izpolni vertikalen ali horizontalen prostor glede na njeno vrednost. V kolikor bi npr. oba `TextView` elementa v horizontalni orientaciji imela parameter `android:weight="0.5"`, potem bi vsak zasedel polovico širine zaslona.

Druga vrsta razporeda prikazuje otroke v relativnih pozicijah. Vsak ima parameter, s katerim povemo, kje se nahaja glede na prejšnjega ali naslednjega otroka ali relativno glede na korenski element (levo, desno ali odspodaj). Pri izdelavi našega projekta je bil zelo priročen, saj smo se lahko izognili gnezdenim razporedom, ki so jih povzročali `LinearLayout`-i. S tem pridobimo lepši izgled *xml* kode in tudi večje performanse ob nalaganju uporabniškega vmesnika.

Otroci obeh razporeditev so bili elementi, s katerimi uporabnik manipulira na zaslonu. Mi smo najbolj pogosto GUI gradili z naslednjimi:

- `Spinner` - izvlečni seznam,
- `TextView` - uporabljali smo za naslove ali opise,
- `EditText` - vnos teksta,
- `ImageButton` - gumb s sliko,
- `ImageView` - slika,
- `AutoCompleteTextView` - hibrid `Spinner`-a ter `EditText`-a.

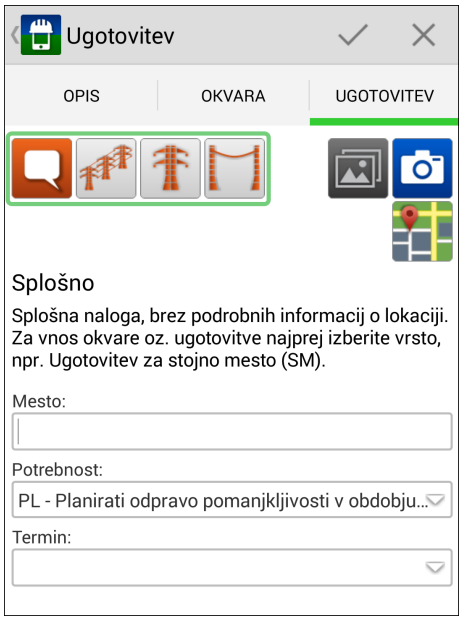
Spodnji primer s sliko bo prikazal kako smo element `ImageView` razporejali znotraj layout-a.

```
<LinearLayout
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_alignParentLeft="true"
    android:orientation="horizontal" >

    <ImageButton
        android:id="@+id/btnConclusionFragSplosno"
        style="@style/ImageButtonWithoutPadding"
        android:layout_width="wrap_content"
        android:layout_height="fill_parent"
        android:background="@drawable/button_effect_splosno" />
    ...
```

Koda 11: Horizontalna razporeditev grafičnih elementov - 1. del

```
        ...
    <ImageButton
        android:id="@+id/btnConclusionFragDaljnovod"
        style="@style/ImageButtonWithoutPadding"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:background="@drawable/button_effect_dv" />

    <ImageButton
        android:id="@+id/btnConclusionFragStojnoMesto"
        style="@style/ImageButtonWithoutPadding"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:background="@drawable/button_effect_sm" />

    <ImageButton
        android:id="@+id/btnConclusionFragRazpetina"
        style="@style/ImageButtonWithoutPadding"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:background="@drawable/button_effect_razpetina" />
</LinearLayout>
```

Koda 12: Horizontalna razporeditev grafičnih elementov - 2. del.



Slika 3.1: Rezultat v zavihku *Ugotovitev* (zeleno obarvano).

3.3 Testne naprave

Na začetku projekta še niso bile določene ciljne naprave, zato smo si ustvarili virtualne naprave, na katerih smo zaganjali aplikacijo. Android SDK vsebuje t. i. *emulator* - virtualno mobilno napravo, ki se zaganja na računalniku. Omogoča razvoj in testiranje Android aplikacij brez uporabe fizične naprave [8]. Pri emulatorjih se je izkazalo, da njihova uporaba bistveno upočasnjuje razvoj programske rešitve. Samo nalaganje virtualne naprave traja tudi do 10 minut, nalaganje aplikacije pa do ene minute. V kolikor aplikacija uporablja še dodatne knjižnice ali pa aplikacijo *debugiramo*[1], potem je postopek nalaganja daljši in onemogoča hiter in normalni razvoj/razhroščevanje.

Po svetovanjih s našimi razvijalci se je stranka odločila za naslednje naprave, ki jih bodo uporabljali njihovi zaposleni:

- tablični računalnik Samsung Galaxy Note 8.0 [18],
- tablični računalnik Samsung Galaxy Tab 3 10.1 [19] in
- mobilni telefon Prestigio 5.0 [20].

Na vseh naštetih napravah smo razvijali in testirali programsko rešitev, grafični vmesnik (oz. *xml* layout) smo morali za vsako napravo posebej prilagajati, saj se je velikost zaslona med njimi razlikovala. Logika je pa za vse ostala ista. Razvoj in razhroščevanje je bilo za razliko od emulatorjev bistveno hitrejše. Povprečni čas nalaganja aplikacije ob razhroščevanju je bil 15 sekund.

USB debugging mode in gonilniki

Za razvoj aplikacije na fizičnih napravah je treba omogočiti t. i. *USB debbuging mode*, ki se nahaja v nastavitvah OS Android. Vsaka naprava je zahtevala tudi USB gonilnik, ki smo ga lahko dobili s spletne strani proizvajalca naprave.

Galaxy Note 8.0 in 10.0

To sta platformi, za kateri se je ob testiranju aplikacije odločilo največ zaposlenih pri stranki. Izkazalo se je, da sta tablična računalnika najboljša za delo na odprtem

[1] Debbuging ali razhroščevanje je metodološki proces, v katerem iščemo in zmanjšujemo število napak ali defekte v neki aplikaciji.

prostoru, saj nudita ogromno prostora na zaslonu za enostavni vnos ugotovitev in ND. Na obeh napravah smo implementirali tudi *master-detail* pogled, ki istočasno prikazuje hierarhično drevo nalogov za delo na desni ter njegove podrobnosti na levi strani zaslona. Izkazalo se je, da je dodajanje ugotovitve ali ND v takem pogledu bistveno hitrejše kot na navadnih 5-inčnih mobilnih telefonih.

Prestigio 5.0

Za razliko od tabličnih računalnikov je bil mobilni telefon najmanj priljubljena naprava. Glavni razlog je predvsem premajhna velikost zaslona, ki je včasih delala probleme ob dodajanju ND ali ugotovitve. Pogled *master-detail* za to napravo ni bil implementiran.

3.4 Splošno o projektu PSAAndroid

Projekt PSAAndroid je bil na začetku glede strukture zelo enostaven. Toda po skoraj enem letu razvoja je postal najbolj kompleksen. Zaradi tega je bilo pri samem načrtovanju določeno, da bo projekt imel dodatne knjižnice z ločeno logiko. PSAAndroid kot tak ima v sebi logiko, ki je namenjena le za Android UI. Npr. logika izmenjave s strežnikom je strogo ločena in se nahaja v knjižnici **LibPsaComData**, vse *utility* metode pa imamo v knjižnici *LibUtil*. Tako v njej hranimo razrede in metode, ki se v glavnem projektu lahko ponavljajo večkrat. Za dostop do persistence pa skrbi logika, ki se nahaja v knjižnici *LibPsaData*.

Razred `DataUtil`, ki je del projekta PSAAndroid, je namenjen pisanju statičnih metod, v katerih se nahaja logika, vezana za knjižnico LibPsaData ter LibPsaComData. Po navadi je šlo za dostop do podatkov, ki smo jih najprej shranili v *ArrayList* in jih nato posredovali v npr. izvlečni seznam oziroma `Spinner`. Za ta razred lahko rečemo, da je *most* med kodo v knjižnicah in glavnem projektu. Spodnji primer (Koda 13.) nam bo jasno razložil potek pridobivanja podatkov ter prikaz le-teh v pogledu `Spinner`. Cilj te metode je v uporabniškem vmesniku prikazati seznam daljnovodov. Za to potrebujemo torej dostop do logike iz knjižnice. V prvi vrstici inicializiramo podatkovno strukturo `dvItems`, ki jo kasneje dodamo v adapter izvlečnega seznama. V drugi vrstici pokličemo statično metodo `getMgDvItems`, ki vrne ArrayList objektov tipa `Mg_dv`. S *foreach* zanko nato napolnimo seznam

`dvItems` in s pomočjo metode `setAdapter()` prikažemo pridobljene podatke v GUI-ju.

```
private void populateSpinnerDV() {
 List<Item<Mg_dv>> dvItems = new ArrayList<Item<Mg_dv>>();
 List<Mg_dv> items = DataUtil.getMgDvItems();
 for (Mg_dv dv : items) {
  dvItems.add(new Item<Mg_dv>(dv, dv.toStr()));
 }
 ArrayAdapter<Item<Mg_dv>> adapter =
  new ArrayAdapter<Item<Mg_dv>>(this, spinner_item, dvItems);
 spinnerDV.setAdapter(adapter);
}
```

Koda 13: Primer pridobivanja podatkov iz objekta *DataUtil*.

Vse vire držimo v za to predvidenih mapah, ki jih zgenerira Eclipse ob ustvarjanju novega projekta. Vse slike so bile v *PNG* obliki, saj smo potrebovali prosojnost ozadja zaradi lepšega videza uporabniškega vmesnika.

3.4.1 Odvisnost projekta in knjižnic

Pri začetnem načrtovanju smo bili pozorni na medsebojno odvisnost glavnega projekta in pripadajočih knjižnic, na koncu smo odvisnosti določili na naslednji način:

- PsaAndroid - reference na LibPsaComData, LibPsaData, LibPsaGis in LibUtil,
- LibPsaComData - referenca na LibUtil,
- LibPsaData - referenca na LibPsaComData in LibUtil,
- LibPsaGis - referenca na LibUtil in
- LibUtil - nima referenc.

Pozorni smo bili tudi na t. i. *circular dependency* [15] ali krožno odvisnost med knjižnicami. Taka napaka bi se lahko zgodila, v kolikor bi bil npr. PsaAndroid odvisen od knjižnice LibUtil in če bi bil LibUtil odvisen od PsaAndroid-a.

3.5 Kontekst PsaApplication

Kontekst je mesto, ki vsebuje globalne informacije o aplikacijskem okolju. Je abstraktni razred OS Androida, ki omogoča pristop do aplikacijskih virov in nje-

govih razredov [10]. Do konteksta dostopamo znotraj aktivnosti s pomočjo metode `getApplicationContext`, ki vrača objekt tipa `Context`. Nalaganje virov, zagon novih aktivnosti, pridobivanje sistemskih servisov in internalnih datotečnih bližnjic - za vse to je potrebna uporaba konteksta. Lahko se pojavlja v različnih instancah, odvisno od tega, katero Androidovo komponento nameravamo uporabljati:

- `Application` - singleton instanca[2], ki se izvaja v življenjskem ciklu aplikacije,
- `Activity/Service` - pojavljata se, kadarkoli *framework* ustvari novo aktivnost ali instanco storitve. Vsak `Activity` ali `Service` ima svojo unikatno instanco,
- `BroadcastReceiver` - ni konkretno kontektst, toda *framework* posreduje le-tega v metodo `onReceive()` od `BroadcastReceiver`-ja,
- `ContentProvider` - tudi ni kontekst, lahko pa dostopamo do njega s pomočjo metode `getContext`, ko se ustvari njegova instanca [11].

Ko se sprašujemo glede življenjske dobe konteksta, je treba vedeti, da le-ta živi skozi celotno življenjsko dobo aplikacije. Ker je globalen, je dostopen torej vsem aktivnostim v območju določene aplikacije.

Naš program je uporabljal predvsem `Application`, ki smo ga razširili v `PsaApplication`. Za pridobitev iz določene aktivnosti smo v tem objektu ustvarili statično metodo `PsaApplication.get(Context)`(Koda 14.). Kot argument pa posredujemo kontekst trenutne aktivnosti. V razredu `PsaApplication` imamo tako splošne nastavitve programa, nastavitve zemljevida in GIS-a (lokacija slojev na napravi), servis za GPS sledenje ter centralni in najbolj pomemben del aplikacije - persistenco oz. shranjevanje vnesenih podatkov.

[2]V objektno-orientiranem programiranju je singleton razred, ki ima le eno instanco. Tako v naši rešitvi obstaja le ena instanca objekta PsaApplication, ki se kreira ob zagonu aplikacije.

```
PsaApplication app;
//LaunchActivity
@Override
protected void onCreate(Bundle savedInstanceState) {
 super.onCreate(savedInstanceState);
 setContentView(R.layout.launch_activity);

 app = PsaApplication.get(this);
}
```

Koda 14: Pridobivanje singleton instance *PsaApplication*.

3.6 Postopek inicializacije aplikacije

`LaunchActivity` je prva aktivnost po zagonu aplikacije. Je glavni upravljalec naslednjih aktivnosti:

- `ServerSettingsActivity` - aktivnost za vnos nastavitev za vzpostavitev povezave s strežnikom in GIS strežnikom,
- `AdminActivity` - omogoča izmenjavo podatkov s strežnikom, nalogo sinhronizacije pa izvaja poseben razred `SyncManager`,
- `LoginActivity` - aktivnost, ki zahteva kredence za vstop v zaščiteni del aplikacije; hierarhični seznam oz. `WoListActivity`.

LaunchActivity je aktivnost brez uporabniškega vmesnika, ki takoj preklopi na eno izmed naštetih aktivnosti glede na vhodne pogoje. V nadaljevanju bomo opisali potek prvega starta po inštalaciji aplikacije na napravo. Opisan bo le del njegove kompleksne funkcionalnosti in zaradi tega drugih številnih lastnosti in preverjanj ne bomo razlagali. Ves scenarij se odvija v metodi `onCreate()`, kjer najprej inicializiramo statični objekt `PsaVersion` (Koda 15.). Verzijo aplikacije določa *Version Name* lastnost. Le-ta se nahaja v Android manifestu[3].

Nato se inicializira aplikacijski kontekst, singleton instanca `PsaApplication`, ki je opisana v prejšnjem poglavju. Temu sledi iskanje datoteke, ki vsebuje nastavitve strežnika in GIS-a. Če se aplikacija prvič sproži po inštalaciji, uporabljamo vgrajene (angl. *built in*) nastavitve, saj glavna datoteka PsaSettings trenutno ne

[3]Manifest je zbirka osnovnih lastnosti Android aplikacije.

```
// Metoda onCreate
try {
   PsaVersion.Initialize(this);
 } catch (Exception e) {
   logger.error("Cannot find app version: ", e);
 }
```

Koda 15: Verzija aplikacije.

obstaja. Kasneje jo bomo v `ServerSettingsActivity` ob shranjevanju vnesenih podatkov tudi ustvarili. Obe datoteki sta tipa *JSON* in ju deserializiramo v objekt `PsaSettings`. Prvič ne bomo imeli inicializirano instanco objekta PsaSettings, prav tako ne bodo nastavitve določene (pogoj `!settings.hasAppSettingsDefined()`). To pomeni, da nam se bo odprla prva aktivnost za nastavitve strežnika `ServerSettingsActivity` s pomočjo metode `startActivityForResult` in integer parametrom oziroma kodo zahteve `requestServerSettingsCode` (Koda 16.), saj se kasneje (ob shranitvi nastavitev) vračamo v `LaunchActivity.onActivityResult()` metodo.

```
// Metoda onCreate
if (settings == null || !settings.hasAppSettingsDefined()) {
 Intent intent = new Intent();
 intent.setClass(this, ServerSettingsActivity.class);
 startActivityForResult(intent, requestServerSettingsCode);
}
```

Koda 16: Zagon aktivnosti *Nastavitve strežnika*.

Slika 3.2: Aktivnost *Nastavitve strežnika* v mobilnem telefonu.

Ko izpolnimo zahtevana polja in je evaluacija podatkov uspešna, lahko kliknemo na gumb *Shrani* ali na ikono za shranjevanje v ActionBar meniju. V objekt PsaSettings dodamo vnesene podatke (metoda populateFromControls()), nato isti objekt serializiramo v PsaSettings.json v metodi `storeLocalSettings`. Postopek shranjevanja je uspešno končan in sedaj lahko zapremo aktivnost z rezultatom `RESULT_OK` (Koda 17.). V kolikor bi bil postopek shranjevanja neuspešen (*bug* v kodi), bi bil uporabnik o tem obveščen in bi imel možnost ponovnega shranjevanja podatkov ali zapiranja aktivnosti s posredovanjem rezultata `RESULT_CANCEL`.

```
// Aktivnost ServerSettingsActivity, metoda onClick
case R.id.save:
  populateFromControls();
  storeLocalSettings();
  setResult(RESULT_OK); finish();
```

Koda 17: Shranjevanje nastavitev - gumb *Shrani*.

Sedaj se nahajamo v metodi `LaunchActivity.onActivityResult()`. Najprej preverimo, ali je rezultat iz prejšnje aktivnosti uspešen (`RESULT_OK`), nato preve-

rimo še kodo zahteve `requestServerSettingsCode`, ki smo jo prej omenili. Tako vemo, iz katere aktivnosti smo prišli in katere akcije moramo sprožiti glede na podano kodo zahteve. V našem primeru preverjamo veljavnost baze z metodo `checkDatabase()`, ki lahko v obliki `enum`-a vrne naslednja stanja:

- Determined - podatkovna baza je določena, saj je bila sinhronizacija izvedena,
- Undetermined - podatkovna baza še ni določena, saj ni bila še niti enkrat izvedena sinhronizacija v celoti,
- Incompatible - napaka pri dostopu do podatkovne baze, npr. podatkovni bazi nista kompatibilni.

Baza še ni določena in zato odpremo `LoginActivity` s kodo zahteve `requestLogin` (Koda 18.).

```
// LaunchActivity.onActivityResult()
...
if (resultCode == RESULT_OK) {
  // Uspesen rezultat, lahko izvedemo zeljeno akcijo
  if (requestCode == requestSystemSettings) {
   DatabaseState dbState = checkDatabase();
    if (dbState == DatabaseState.Determined ||
               dbState == DatabaseState.Undetermined) {
     Intent intent = new Intent(this, LoginActivity.class);
     startActivityForResult(intent, requestLogin);
     overridePendingTransition(R.anim.fade_in, R.anim.fade_out);
   } else {
    showDialog("Opozorilo", "Podatkovna baza je neustrezna. Ali
    boste odstranili bazo?", tagRemoveDb);
   }
  }
}
finish();
...
```

Koda 18: Odpremo aktivnost za prijavo.

Če baza ni določena, nam LoginActivity ponudi vpis uporabniškega imena in gesla ter obvezno možnost izmenjave podatkov s strežnikom. Ob pritisku na gumb *Prijava* najprej preverimo, ali imamo vzpostavljeno povezavo s strežnikom in nato validiramo kredence uporabnika `loginToServerAndAuthenticate()`.

V kolikor pa je baza določena, bi naredili lokalni login in bi takoj prišli v hierarhični seznam oz. `WoListActivity`. Če so vnešeni podatki točni, se trenutna aktivnost konča in LaunchActivity ponovno prevzame kontrolo; če pa niso, potem uporabnika opozorimo, da je prišlo do napake ob poskusu logina. Ob zapiranju trenutnega Activity-ja poskrbimo tudi za avtomatično sinhronizacijo, ne da bi uporabnik kliknil na gumb *Sinhronizacija* v `AdminActivity`. Zato v objekt `intent` dodamo parameter `sync`, ki ima vrednost `true` (Koda 19.).

```
boolean loginSuccessful = loginToServerAndAuthenticate()
sync = true; // Izmenjava podatkov
if (!loginSuccessful) {
        showAlertDialog("Opozorilo", getError());
} else {
        intent.putExtra(parSync, sync); finish();
}
```

Koda 19: Prijava na strežnik z gumbom *Prijava*.

Slika 3.3: Aktivnost *login* v mobilnem telefonu.

Iz login aktivnosti smo prišli nazaj v metodo LaunchActivity.onActivityResult(). Ker poznamo naš *request code*, ki smo ga poslali pred odpiranjem `LoginActivity`-ja, pademo v vrstico kode, ki takoj sproži novo aktivnost `AdminActivity` oziroma njen fragment `AdminComFragment`. Spomnimo se sedaj poslane boolean vrednosti `sync` iz prejšnje aktivnosti. Za pridobitev potrebujemo objekt `Intent`, ki je zadolžen za prenos tovrstnih podatkov pri medsebojni komunikaciji med aktivnostmi. Le-ta je vključen v metodi onActivityResult() kot parameter. Za pridobitev pokličemo metodo `getBooleanExtra(LoginActivity.parSync, false)`. Prvi argument nam služi kot ključ za dostop do vrednosti, drugi pa kot privzeta vrednost, v kolikor je Intent ne vsebuje. Ampak: mi se že prej vprašamo, ali *intent* sploh vsebuje ta ključ, in sicer z metodo `data.hasExtra(LoginActivity.parSync)`. V kolikor je *sync* vrednost postavljena, potem jo dodamo v nov `Intent` (`intent.putExtra(AdminActivity.parExchange, true)`). Aktivnost zaženemo s pomočjo rezultata oz. kode zahteve `requestAdmin`. Če vrednost ni postavljena, potem zaženemo metodo `openWoListActivity()`, ki odpre aktivnost s hierarhičnim seznamom (Koda 20.).

```
// LaunchActivity
@Override
protected void onActivityResult(int requestCode,...,Intent data){
...
else if (requestCode == requestLogin) {
  boolean sync = data.hasExtra(LoginActivity.parSync) &&
  data.getBooleanExtra(LoginActivity.parSync, false);
  if (sync) {
    Intent intent = new Intent();
    if (sync) {
          intent.putExtra(AdminActivity.parExchange, true);
    }
    intent.setClass(this, AdminActivity.class);
    startActivityForResult(intent, requestAdmin);
    finish();
} else {
    openWoListActivity();
}
}
...
}
```

Koda 20: Zagon aktivnosti za sinhronizacijo s pomočjo *LaunchActivity*.

Z zagonom AdminActivity-ja smo ponovno začasno prekinili izvajanje LaunchActivity-ja. Sedaj se nahajamo v `AdminComFragment`-u, saj je `AdminActivity` napisan tako, da v sebi poleg omenjenega fragmenta vsebuje še 3, in sicer:

- `AdminInfoFragment` - Podatki o uporabniku,
- `AdminDbRestoreFragment` - Obnovitev podatkovne baze in
- `AdminDbWipeFragment` - Brisanje podatkovne baze.

Vsi štirije fragmenti so povezani v ViewPager-ju. Med izmenjavo onemogočimo dostop do ostalih, saj je odstranjevanje ali obnavljanje podatkovne baze v tem trenutku nedovoljeno. V metodi `AdminComFragment.onResume()` zaženemo avtomatsko izmenjavo s strežnikom (Koda 21.). Ob koncu sinhronizacije se na

```
// AdminComFragment
public void onResume() {
 Intent intent = getActivity().getIntent();
 boolean sync= intent.hasExtra(AdminActivity.parExchange) &&
   intent.getBooleanExtra(AdminActivity.parExchange, false);
 if (sync) {
    synchronize();
 }
}
```

Koda 21: Zagon sinhronizacije v fragmentu *AdminComFragment*.

kratko vrnemo v `LaunchActivity.onActivityResult()`. Vemo, da smo prišli iz `AdminActivity`-ja (torej spremenljivka `requestAdmin`), zato za konec pokličemo metodo `openWoListActivity()`, ki uspešno konča LaunchActivity (ne pošiljamo več rezultata naprej) in odpre hierarhični seznam `WoListActivity`:

```
// LaunchActivity - onActivityResult metoda
else if (requestCode == requestAdmin) {
 DatabaseState dbState = checkDatabase();
 if (dbState == DatabaseState.Determined) {
   openWoListActivity();
 } else {
    showDialog("Napaka", "Izvedi sinhronizacijo!", tagSync);
 }
}
```

Koda 22: Zaključek inicializacije.

3.7 Persistenca

PSAAndroid uporablja [9] persistentnčni[4] sistem za hrambo vseh podatkov vnesenih s strani uporabnika. Razred namenjen shranjevanju podatkov se imenuje `PsaDataEngine` in je eden glavnih delov projekta, ki omogoča nemoteno delovanje aplikacije. `PsaDataEngine` je singleton instanca. Ob prvem zagonu aplikacije pa je dostopen šele, ko so nastavitve (serializiran objekt `PsaSettings`) preverjene in urejene. Do njega dostopamo preko aplikacijskega konteksta. V spodnji kodi je opisan primer dostopanja do instance iz aktivnosti:

```
@Override
public PsaDataEngine getDataEngine() {
 return PsaApplication.get(this).getDataEngine();
}
```

Koda 23: Persistenčni objekt *PsaDataEngine*.

Shranjevanje in brisanje

Shranjevanje in brisanje podatkov bomo razložili na primeru vnašanja ugotovitve ali naloga za delo.

Aktivnost za dodajanje ugotovitve (`OrderDetailActivity`) vsebuje skupni objekt `OrderDetailContext`, v katerem hranimo vnesene podatke s strani uporabnika. V kolikor uporabnik obrne zaslon ali zamenja fragment (npr. `ViewPager` s fragmenti), se pred temi dogodki pokliče metoda, ki vse vnesene podatke shrani v omenjeno instanco objekta. Vendar se podatki še ne zapišejo v persistenco, vse dokler ne pretisnemo na gumb *Shrani*. Logika preveri še, ali so vsi podatki pravilno vneseni. Če so, sledi proces shranjevanja podatkov v persistentno obliko s pomočjo metode `saveToDataEngine`, ki se nahaja v omenjenem `OrderDetailActivity`-ju.

Začnemo s shranjevanjem geo podatkov za zemljevid, v kolikor smo ob dodajanju ugotovitve izbrali stojno mesto ali razpetino kot njen tip. Nato smo pridobili `OrderDetailContext`, v katerem se nahajata dva glavna objekta `Mx_workorder_v`

[4]Persistenca se s stališča računalništva nanaša na karakteristično stanje, ki živi izven procesa in je ustvarjen s njegove strani.

in `Mm_fieldtask` (v njiju smo dodajali podatke iz uporabniškega vmesnika). Iz objekta `fieldtask` smo pobrisali nepotrebne podatke, saj ob shranjevanju potrebujemo vnesene podatke za točno izbran tip ugotovitve. Drugi (nepotrebni) podatki bi v nasprotnem primeru lahko povzročili težave ob sinhronizaciji. Prvi del metode `saveToDataEngine()` lahko vidimo v spodnji kodi:

```
public void saveToDataEngine() {
saveGeoData(); // Shranimo podatke za zemljevid

OrderDetailContext odc = getDetailContext();

Mx_workorder_v workorder = odc.getWorkorder();
Mm_fieldtask fieldtask = odc.getFieldtask();

if (fieldtask != null && fieldtask.getEvidencetype() != null) {
        if (fieldtask.isEvidenceType_A()) {
                fieldtask.setDvid(null);
                fieldtask.setSmid(null);
                fieldtask.setSm2id(null);
                fieldtask.setSpanid(null);
        } else if (fieldtask.isEvidenceType_DV()) {
                fieldtask.setSmid(null);
                fieldtask.setSm2id(null);
                fieldtask.setSpanid(null);
        } else if (fieldtask.isEvidenceType_SM()) {
                fieldtask.setSm2id(null);
                fieldtask.setSpanid(null);
        }
}
...
```

Koda 24: Shranjevanje v persistenco - 1. del.

Objekt `fieldtask` je sedaj pripravljen s podatki izbranega tipa ugotovitve. Nadaljujemo s pripravljanjem instance `dataEngine` za postopek zapisovanja v podatkovno bazo. Pred tem moramo določiti t. i. `wonum` ter `fieldTaskId` - identifikacijsko številko naloga za delo in ugotovitve. Številka je namreč na začetku nedoločena (vrednost `null`), v kolikor dodajamo nov nalog za delo. `wonum` bo dobil novo generirano 6-mestno številko s prefiksom (npr. *PSAM000101*), `fieldTaskId` pa negativno številko (npr. *-101*). Na takšen način bo logika sinhronizacije s strežnikom lahko razlikovala med novododanimi nalogi za delo ter tistimi, ki so bili na napravo naloženi ob prejšnji sinhronizaciji (le-ti imajo 6 mestno številko brez prefiksa *PSAM*). Če je identifikacijska številka določena

(`wonum` je različen od `null`), potem se generiranje nove preskoči. Zadnji korak je uporaba metode `dataEngine.store(workorder)`, ki v datoteko shrani vsebino objekta `Mx_workorder_v` in `Mm_fieldtask` (Koda 25.).

```
...
// Nadaljevanje metode saveToDataEngine()

PsaDataEngine dataEngine = getDataEngine();
if (workorder.getWonum() == null) {
        try {
                int wonum = dataEngine.getNextId();
                workorder.mobi_GenerateWonum(wonum);
                fieldtask.setWonum(workorder.getWonum());
                int ftId = -dataEngine.getNextId();
                fieldtask.setFieldtaskid(ftId);
        } catch (Exception e) {
                logger.error("saveToDataEngine: ", e);
        }
}

try {
        dataEngine.store(workorder);
        if (fieldtask != null) {
                getDataEngine().store(fieldtask);
        }
} catch (PsaDataPersisterException e) {
        logger.error("saveToDataEngine: ", e);
}

getActivity().setResult(Activity.RESULT_OK);
getActivity().finish();
```

Koda 25: Shranjevanje v persistenco - 2. del.

Brisanje ugotovitve je bolj enostavno kot samo dodajanje. V PSA projektu uporabnik lahko briše le ugotovitve ali naloge za delo, ki jih je sam dodal (v hierarhičnem seznamu oranžna ikona). ND ali ugotovitve, ki jo dobimo ob sinhronizaciji s strežnikom, pa ni možno odstranjevati (v hierarhičnem seznamu imajo modro ikono). Uporabnik torej lahko izbere ND ali ugotovitev z oranžno barvo, kjer se odpre novo okno (`OrderDetailActivity`), v katerem v izbirnem meniju pritisne ikono *Odstrani* (oblike koša za smeti). V tem trenutku sprožimo metodo `deleteFromDataEngine` (Koda 26.). Najprej odstranimo morebitne geopodatke s pomočjo metode `deleteGeoData()`, nato pridobimo instanci `workorder` in `fieldtask` iz začasnega objekta za hrambo podatkov `OrderDetailContext`. V

kolikor imata `workorder` in `fieldtask` status dodanih objektov (*flag* določimo ob dodajanju nove ugotovitve ali ND) potem ju lahko brišemo iz persistence oz. ustreznih datotek. Za to poskrbi ukaz `delete()` iz objekta `PsaDataEngine`. Brišemo tudi vse posnete slike z metodo `removeAllPictureFilesForWonum`, ki kot argument dobi *id* številko naloga za delo. Na koncu še zaključimo `OrderDetailActivity` in se vrnemo v hierarhični seznam.

```
public void deleteFromDataEngine() {
deleteGeoData();

Mx_workorder_v workorder = getDetailContext().getWorkorder();
Mm_fieldtask fieldtask = getDetailContext().getFieldtask();
boolean woIsAdded = workorder.getIsadded();
boolean ftIsAdded = fieldtask.getIsadded();

if (woIsAdded && ftIsAdded) {
 try {
   getDataEngine().delete(fieldtask);
   getDataEngine().delete(workorder);
   if (deletePhotos) {
        PsaDataEngine engine = getDataEngine();
        engine.getSettingsFiles().
                removeAllPictureFilesForWonum(workorder.getWonum());
   }
 } catch (PsaDataPersisterException e) {
        logger.error("onOptionsItemSelected: ", e);
 } catch (PsaSettingsException e) {
        logger.error("onOptionsItemSelected: ", e);
 }
   } else if (!woIsAdded && !ftIsAdded) {
        logger.warn("Ni mozno brisati!");
   }
 getActivity().setResult(RESULT_WORKORDER_DELETED);
 getActivity().finish();
}
```

Koda 26: Brisanje iz persistence.

Lokacija baze

Android ima posebno datotečno strukturo, namenjeno le za odlaganje aplikacijskih podatkov, ki jo ob inštalaciji aplikacije ustvari na osnovi *namespace*-a[5] aplikacije.

[5]Namespace je kontekst identifikatorja za nek programski jezik, kot je npr. mapa ali njene podmape.

Poleg tega ustvari še mapo *files*. Ob deinštalaciji se vsi podatki, vključno z direktoriji aplikacije, avtomatično izbrišejo. Ker aplikacija omogoča izdelavo varnostne kopije podatkov, smo ob procesu backup-a podatke prenesli v korensko mapo primarnega eksternega pomnilnika. V njej so poleg varnostnih kopij tudi *JSON* datoteke za razhroščevanje in fotografije, posnete ob dodajanju nove ugotovitve. V tem delu se omenjene datoteke ne bodo brisale ob deinštalaciji. Tako bodo npr. ob naslednji inštalaciji isti podatki na voljo in iz njih bomo lahko obnovili podatkovno bazo. Pot do aplikacijskih podatkov oz. baze v našem primeru izgleda takole:

- `\mnt\sdcard\Android\psa\miga\eu\mobile\files`.

Podatke hranimo v tekstovni datoteki s končnico *.db*, ki se na napravi nahaja v primarnem eksternem pomnilniku, kjer se nahajajo naša podatkovna baza, JSON nastavitve strežnika ter geopodatkovna baza. Podatkovno in geobazo pa pridobimo s pomočjo sinhronizacije iz strežnika.

3.8 Podrobnosti naloga za delo

Po uspešni prijavi v aplikacijo se nahajamo v hierarhičnem seznamu (`WoList Activity`). Na vrhu seznama je označen prvi nalog za delo. Lahko se odločimo za dodajanje novega naloga za delo in kliknemo na + ikono v meniju na desni strani zgoraj ali kliknemo na modro ikono v izbrani vrstici (pregled obstoječega ND). WoListActivity z rezultatom sproži aktivnost `OrderDetailActivity`. Ta aktivnosti je vsebovana iz več zavihkov (v nadaljevanju fragmentov). Prvi zavihek *Opis* služi za opis okvare, drugi zavihek *Okvara* je za hierarhično klasificiranje napak. Zadnji zavihek *Ugotovitev* je namenjen za vnos ugotovitve, ki je lahko povezana z daljnovodom, stojnim mestom ali razpetino. Omogoča prikaz daljnovodov in stojnih mest na zemljevidu (`MapActivity`) in tudi pregled tehničnih podatkov (`TechDetailActivity`). Vsak zavihek omogoča snemanje in pregled slik, ki se shranijo v podatkovno bazo s pritiskom na gumb `Shrani`, in sicer na desni strani v ActionBar meniju.

Da bi omogočili prikaz zavihkov, je bilo treba implementirati več različnih razredov, obenem smo morali tudi obvladati fragmente ter razumeti njihov življenjski cikel. Glavna naloga je bila torej izdelava UI-ja, pravilna implementacija Androidovega pogleda `ViewPager` s pripadajočimi fragmenti in povezava UI-ja s podatkovnim modelom (iz objekta `PsaDataEngine`). `OrderDetailActivity` je imel po enem mesecu razvoja naslednjo strukturo:

- `OrderDetailFragment` - fragment, ki poišče objekt `Mx_workorder_v` in `Mm_fieldtask` iz podatkovnega modela in ju inicializira. V kolikor tam ne obstajata (dodajamo novo ugotovitev), potem ju inicializiramo s privzetimi nastavitvami,
- `OrderDetailContext` - vsebuje objekta `Mx_workorder_v` in `Mm_field task` za hrambo stanja vnesenih podatkov ob pritisku iz enega v drug zavihek,
- `OrderDetailPagerAdapter` - v njem naredimo instance fragmentov, ki bodo prikazani kot zavihki v GUI-ju. Trenutno imamo 3 različne fragmente:
  - `OrderDetailDescriptionFragment` - Zavihek *Opis*,
  - `OrderDetailFailureFragment` - Zavihek *Okvara* in
  - `OrderDetailConclusionFragment` - Zavihek *Ugotovitev*.

`ViewPager` je Androidova kontrola, ki omogoča horizontalno drsanje v levo ali desno smer oz. prikaz zavihkov. Za njegovo delo je pred tem potrebno definirati `OrderDetailPagerAdapter`, ki pa vsebuje potrebne instance fragmentov (oz. zavihkov). Torej adapter lahko manipulira z instancami (lahko jih npr. skriva) in se obnaša kod podatkovni model, ViewPager pa uporabniku omogoča drsanje po zaslonu. Ustvarjanje obeh instanc se izvaja v `OrderDetailFragment.onCreateView()` metodi (Koda 27.).

```
// OrderDetailFragment
@Override
public View onCreateView(LayoutInflater inflater, ViewGroup view,
                                        Bundle savedInstanceState) {
 // Priredimo instanco adapterja, ki bo ustvaril
 // tri instance fragmentov (zavihkov)
 OrderDetailPagerAdapter detailPagerAdapter =
   new OrderDetailPagerAdapter(this.getChildFragmentManager(), 3);

 // Nato instanco adapterja posredujemo kot argument
 // metode setAdapter
 ViewPager viewPager = (ViewPager) view.findViewById(R.id.pagerDetail);
 viewPager.setAdapter(detailPagerAdapter);
 viewPager.setOnPageChangeListener(this);
 return rootView;
}
```

Koda 27: Inicializacija *ViewPager*-ja in *OrderDetailPagerAdapter*-ja.

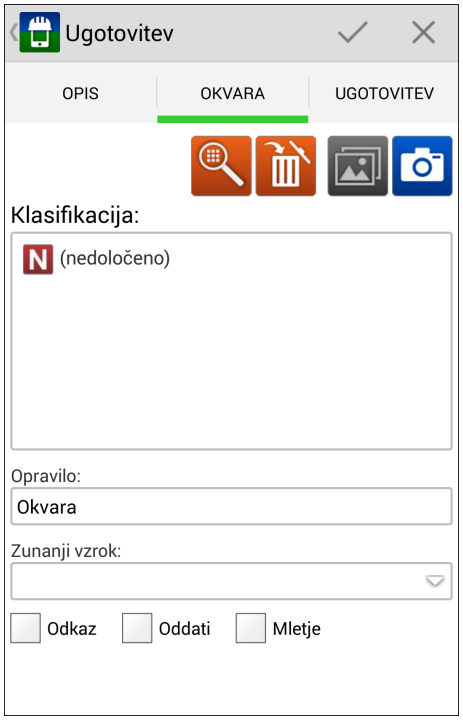
`OrderDetailContext` je razred, ki vsebuje vse potrebne objekte iz baze za prikaz podatkov v uporabniškem vmesniku. Dva glavna sta `Mx_workorder_v` ter `Mm_fieldtask`, shranili ju bomo pa v persistenco z metodo `OrderDetail Fragment.saveToDataEngine()`. Vsak zavihek vsebuje metodi `populateTo Controls()` (bere iz objekta `OrderDetailContext` in jih prikaže v GUI-ju) in `populateFromControls` (piše v objekt `OrderDetailContext`). Prva se pokliče ob nalaganju posameznega fragmenta ali prehodu iz enega v drugi zavihek. Drugo metodo pokličemo tudi ob prehodu iz enega v drug zavihek ter pred ponovnim zagonom aktivnosti, ob ponovnem nalaganju jih pridobimo s pomočjo serializacije objekta OrderDetailContext. V spodnji kodi bomo prikazali uporabo obeh metod v zavihku `OrderDetailFailureFragment` (GUI je na Sliki 3.4.).

```
 // OrderDetailFailureFragment
private void populateToControls() {
 Mx_workorder_v workorder = getDetailContext().getWorkorder();

 // Opravilo
 edtFailureFragTitle.setText(workorder.toDescriptionOrP());

 // Odkaz, Oddati, Mletje
 chkFailureFragOdkaz.setChecked(workorder.getOdkaz());
 chkFailureFragToSubmit.setChecked(workorder.getOddati());
 chkFailureFragMilling.setChecked(workorder.getMletje());

}
```

Koda 28: Metoda *populateToControls()*.

```
 // OrderDetailFailureFragment
private void populateFromControls() {

 if (getDetailContext() != null && edtFailureFragTitle != null) {
   Mx_workorder_v workorder = getDetailContext().getWorkorder();

   // Opravilo
   String desc = edtFailureFragTitle.getText().toString();
   workorder.updateDescriptionOrP(desc);

   // Odkaz, Oddati, Mletje
   workorder.setOdkaz(chkFailureFragOdkaz.isChecked());
   workorder.setOddati(chkFailureFragToSubmit.isChecked());
   workorder.setMletje(chkFailureFragMilling.isChecked());

 }

}
```

Koda 29: Metoda *populateFromControls()*.



Slika 3.4: Zavihek *Okvara*.

Poglavje 4

Predstavitev končne rešitve

Končna rešitev vsebuje veliko funkcionalnosti, ki smo jih razvijali in testirali v času enega leta. V nadaljevanju bomo s pomočjo zaslonskih slik demonstrirali uporabo le določenih in najbolj pogosto uporabljanih. Kot zanimivost lahko končno rešitev opišemo tudi v številkah:

- PsaAndroid (brez pripadajočih knjižnic) je vseboval okoli 16000 vrstic kode,
- čez 100 slik v *png* formatu, ki so se uporabljale kot ikone, gumbi ipd,
- čez 50 različnih *layout*-ov za tablice in mobilne telefone,
- vsebuje 15 *namespace*-ov,
- 30 različnih razredov (fragmenti, aktivnosti in *utility* razredi),
- projekt je velikosti 8MB,
- 1500 *commit*-ov na SVN strežnik.

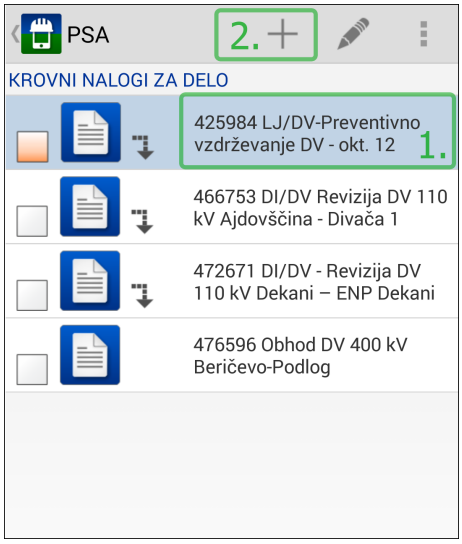
Sam projekt se seveda nadgrajuje glede na zahteve stranke, zato se te številke vsakodnevno spreminjajo.

4.1 Demonstracija uporabe

Vnos nove ugotovitve - Daljnovod

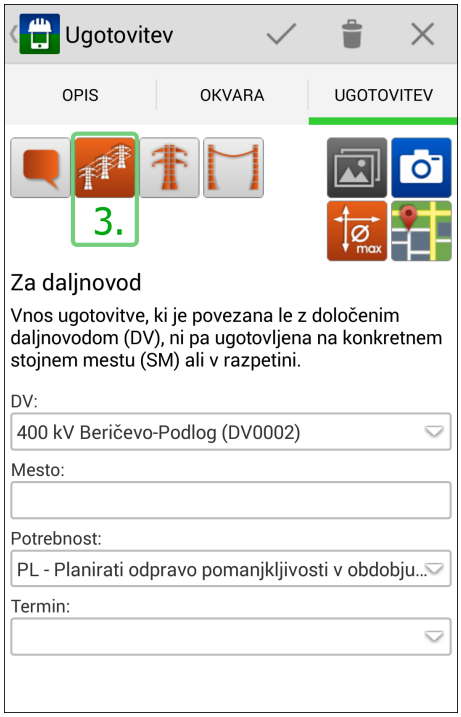
Pri izvajanju obhoda oglednika ali pri izvajanju rednih vzdrževalnih nalog lahko delavec opazi okvaro ali druge vrste ugotovitve in jo evidentira v mobilni napravi.

V seznamu izberemo nalogo, pri kateri smo opazili novo okvaro ali drugo ugotovitev. Vrstico v seznamu izberemo s pritiskom na tekst (1). Za dodajanje nove ugotovitve nato pritisnemo ikono (2), ki odpre okno za vnos nove ugotovitve.



Slika 4.1: V seznamu izbran ND.

Najprej s pritiskom na ikono (3) izberemo vrsto; odločimo se za daljnovod. Daljnovod je vrsta ugotovitve, ki je povezana z določenim daljnovodom, ni pa na konkretnem stojnem mestu ali razpetini. Privzeto je ponujen DV, ki ga določa lokacija naloge oz. ND, pri katerem vnašamo ugotovitev. V kolikor je lokacija povezana z več daljnovodi (npr. lokacija na določenem DS, ki se razteza preko več DV), potem spustni seznam ponuja vse DV, ki so povezani s to lokacijo.

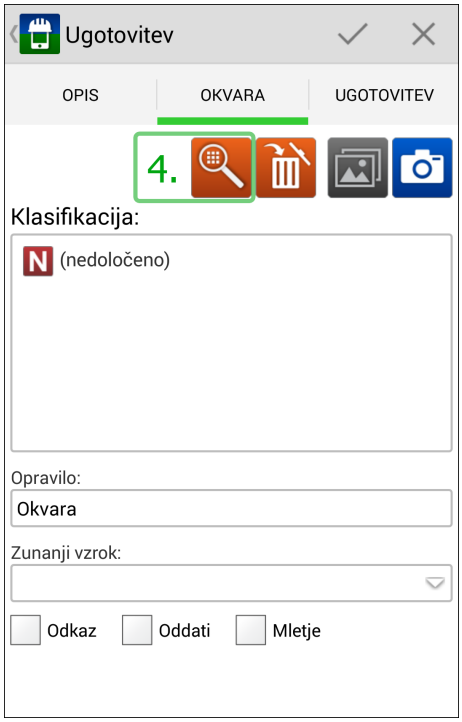


Slika 4.2: Vrsta napake - Daljnovod.

Nato kliknemo na zavihek *Okvara*, kjer opišemo in klasificiramo napako. Klasifikacijo okvare določa šifrant okvar, ki je tipično sestavljen v nivojih od:

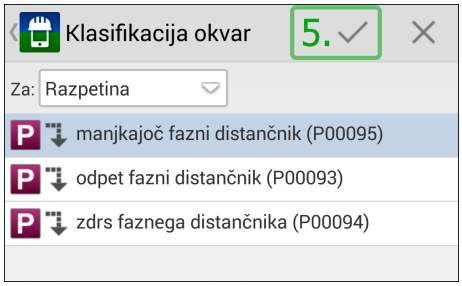
- Napaka,
- Problem,
- Vzrok in
- Odprava.

Naziv okvare je poljuben tekst, ki se v sistemu Maximo prikaže kot opis opravila. *Zunanji vzrok* je neobvezna izbira, ki opiše vzrok nastanka okvare. Odpremo dialog za izbiro klasifikacije v šifrantu okvar (4).



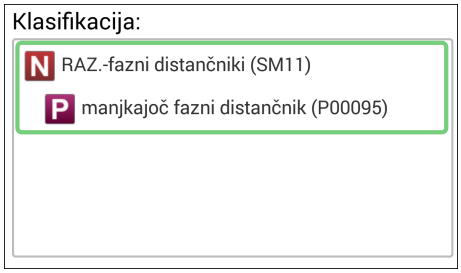
Slika 4.3: Odpremo šifrant okvar.

Klasifikacijo okvare na terenu določimo le do nivoja, do katerega so informacije znane, običajno je znan vsaj problem, lahko pa tudi vzrok, včasih pa je mogoče določiti tudi način odprave. V našem primeru izberemo problem in nato izbiro potrdimo s klikom na ikono (5) v ActionBar meniju.



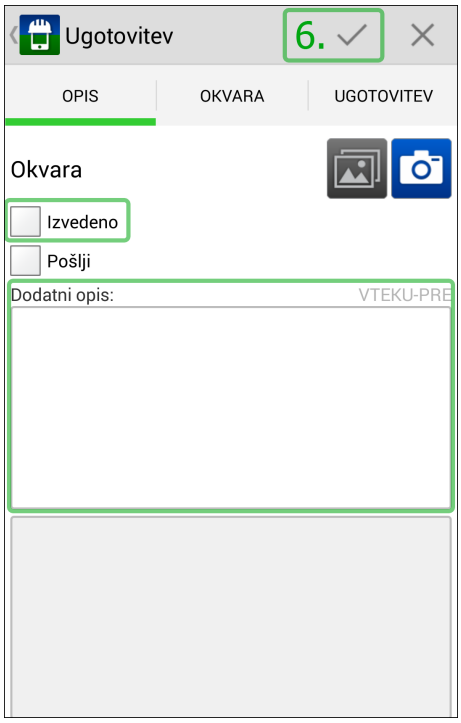
Slika 4.4: Aktivnost *Klasifikacija okvar*.

Klasifikacija okvar nas vrne v zavihek *Okvara*, kjer se nam prikaže klasifikacija do izbranega nivoja *Problem* (zeleno obarvano). Po določitvi opisa okvare nadaljujemo z vnosom v zavihku *Opis*.



Slika 4.5: Osveženi nivojski seznam v zavihku *Okvara*.

V zadnjem koraku neobvezno vnesemo *Dodatni opis*, v kolikor pa smo okvaro že ob ugotovitvi tudi odpravili, pa označimo *Izvedeno*. Z gumbom (6) v ActionBar meniju vnos ugotovitve zaključimo. V kolikor vnaprej vemo, da ne bomo v zavihek Opis ničesar vnesli, potem lahko vnos ugotovitve zaključimo že kar v zavihku *Okvara* ali pa celo že v zavihku *Ugotovitev*, v kolikor vemo, da ne moremo določiti niti klasifikacije napake. Vnesena okvara je nato prikazana v seznamu.



Slika 4.6: Zadnji korak vnosa ugotovitve.

Poglavje 5

Zaključek

V diplomski nalogi smo videli potek izdelave projekta PSAAndroid od temeljev (papirna oblika GUI-ja) pa vse do konkretne implementacije Androidovih tehnologij v končni rešitvi. Za razvijalca začetnika tak projekt pomeni pridobivanje novih izkušenj ter ogromno dodatnega znanja. Med razvojem lahko podrobno spozna pomen *Activity*-ja ter *Fragment*-a, njun življenjski cikel: od metode `onCreate` do metode `onDestroy`. Nauči se tudi timskega dela in razdelitve zahtevanih nalog na majhne podprobleme. Na začetku je težko določati čas za izdelavo določenih nalog, sčasoma pa tudi to postane lažje. Dobro je namreč okvirno vedeti, kdaj bodo določeni problemi rešeni, da se lahko nadaljuje z nemotenim razvojem.

OS Android je dostopen vsem, lahko ga kdorkoli spreminja in prilagaja za svoje namene, kar je glavni razlog takšne popularnosti. OS Android počasi postaja dostopen v vseh življenjskih sferah. Tako je pred kratkim izdal novo verzijo operacijskega sistema (Android Wear [12]), ki omogoča prikaz informacij na majhnih prenosnih napravah, kot so npr. ročne ure. Dostopen bo tudi v avtomobilih kot Android Auto [14] (GPS navigacijski sistem) in v televizijah kot Android TV [13]. Glede na to lahko rečemo, da ima OS Android še veliko potenciala in da bo tudi v prihodnosti glavni na trgu mobilnih operacijskih sistemov.

Literatura

[1] (2013) “Worldwide Mobile Phone Sales”,
Dostopno na: http://www.gartner.com/newsroom/id/2573415L.

[2] (2013) “Getting Started with Android Studio”,
Dostopno na: http://developer.android.com/sdk/installing/
studio.html.

[3] (2013) “Apache License”,
Dostopno na: http://en.wikipedia.org/wiki/Apache_License.

[4] (2014) “Subversion”,
Dostopno na: http://hr.wikipedia.org/wiki/Subversion.

[5] (2014) “Apache Subversion”,
Dostopno na: http://en.wikipedia.org/wiki/Apache_Subversion.

[6] (2014) “Android Fragments”,
Dostopno na: http://developer.android.com/guide/components/
fragments.html.

[7] (2014) “Google Sites”,
Dostopno na: http://learn.googleapps.com/sites.

[8] (2014) “Android Emulator”,
Dostopno na: http://developer.android.com/tools/help/
emulator.html.

[9] (2014) “Persistence (computer science)”,
Dostopno na: http://en.wikipedia.org/wiki/
Persistence_(computer_science).

[10] (2014) “Context”,
Dostopno na: http://developer.android.com/reference/android/content/Context.html.

[11] (2013) “Context, What Context?”,
Dostopno na: http://www.doubleencore.com/2013/06/context/.

[12] (2014) “Android Wear”,
Dostopno na: http://developer.android.com/wear/index.html.

[13] (2014) “Android TV”,
Dostopno na: http://developer.android.com/tv/index.html.

[14] (2014) “Android Auto”,
Dostopno na: http://developer.android.com/auto/index.html.

[15] (2014) “Circular dependency”,
Dostopno na: http://en.wikipedia.org/wiki/Circular_dependency.

[16] (2014) “Linear Layout”,
Dostopno na: http://developer.android.com/reference/android/widget/LinearLayout.html.

[17] (2014) “Relative Layout”,
Dostopno na: http://developer.android.com/guide/topics/ui/layout/relative.html.

[18] (2014) “Samsung Galaxy Note 8.0 Specifications”,
Dostopno na: http://www.samsung.com/global/microsite/galaxynote/note8.0/specifications.html

[19] (2014) “Samsung Galaxy Tab 3 10.1 Specifications”,
Dostopno na: http://www.samsung.com/global/microsite/galaxytab3/spec_10.html

[20] (2014) “MulitPhone 5044 DUO”,
Dostopno na: http://www.prestigio.si/izdelki/Smartphones/MultiPhone/MultiPhone_5000_DUO

Slike